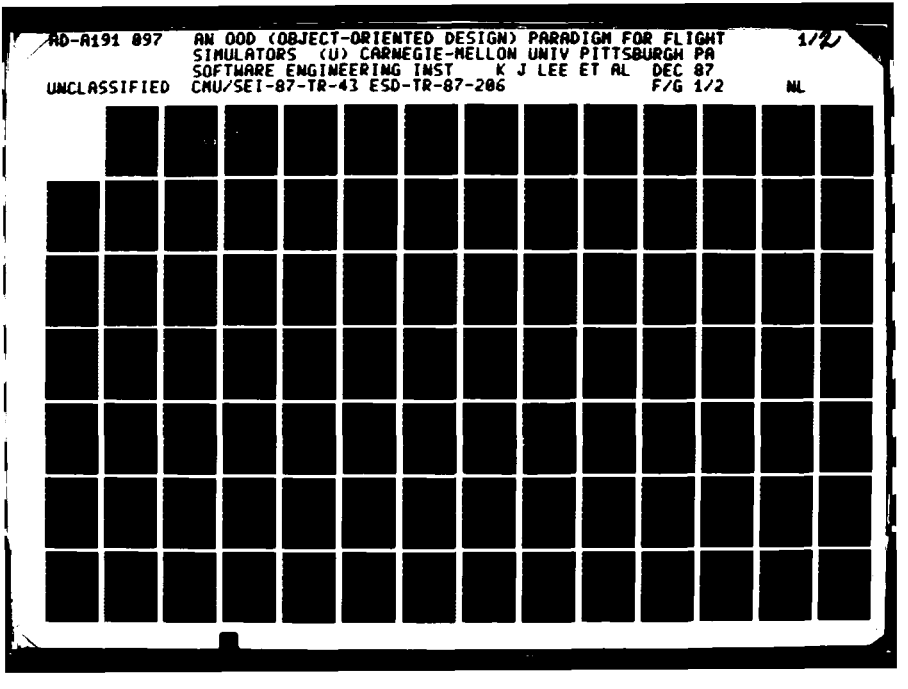
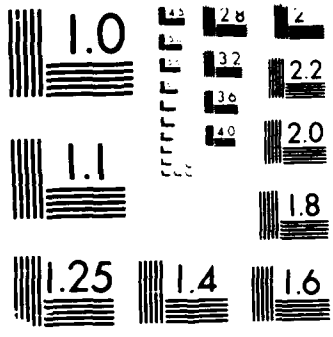

AD-A191 897 AN OOD (OBJECT-ORIENTED DESIGN) PARADIGM FOR FLIGHT SIMULATORS (U) CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST K J LEE ET AL DEC 87
1/2
UNCLASSIFIED CMU/SEI-87-TR-43 ESD-TR-87-206 F/G 1/2 NL



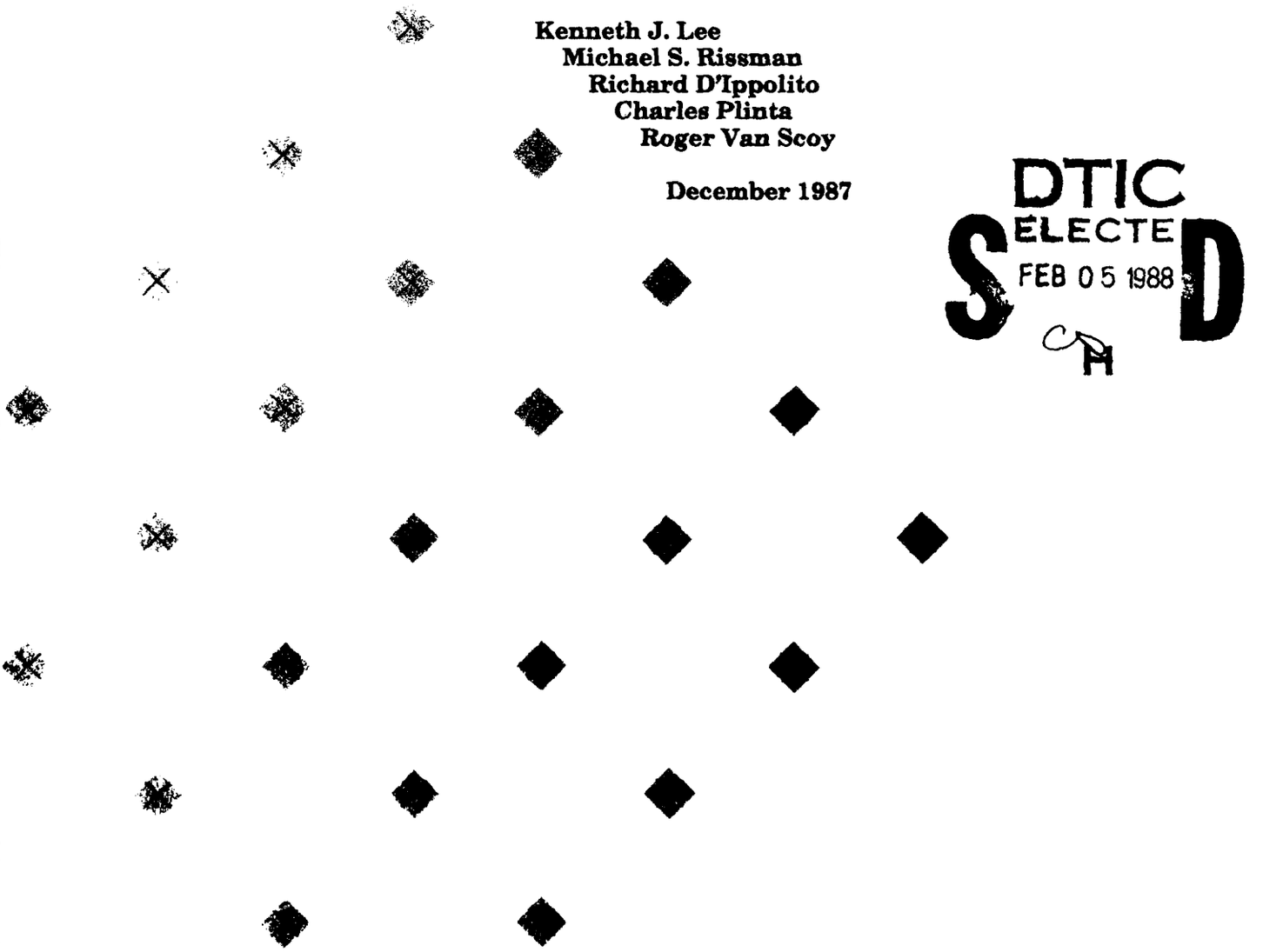
MICROCOPY RESOLUTION TEST CHART

Technical Report
CMU/SEI-87-TR-43
ESD-TR-87-206

An OOD Paradigm for Flight Simulators

Kenneth J. Lee
Michael S. Rissman
Richard D'Ippolito
Charles Plinta
Roger Van Scoy

December 1987

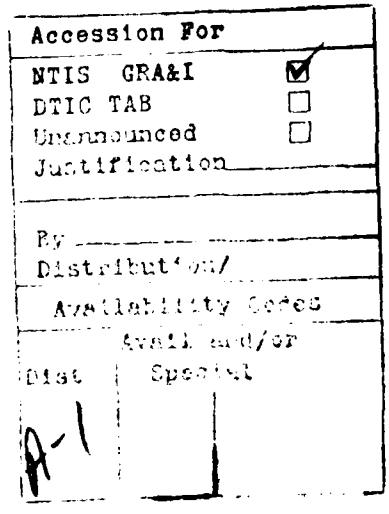


Technical Report
CMU/SEI-87-TR-43
ESD-TR-87-206
December 1987

An OOD Paradigm for Flight Simulators

Kenneth J. Lee
Michael S. Rissman
Richard D'Ippolito
Charles Plinta
Roger Van Scoy

Dissemination of Ada Software Engineering Technology



Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

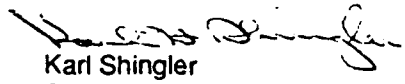
SEI Joint Program Office
ESD/XRS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER



Karl Shingler
SEI Joint Program Office

This work was sponsored by the U.S. Department of Defense.

Table of Contents

List of Figures

1. Introduction

1.1. Background

This report presents a paradigm for object-oriented implementations of flight simulators. It is a result of work on the Ada Simulator Validation Program (ASVP) carried out by members of the technical staff at the Software Engineering Institute (SEI).

1.2. Motivation

Object-oriented design predominates discussions about Ada-based software engineering. The identification of objects and the implementation of objects are two separate issues. This paradigm is a model for implementing systems of objects. The objects are described in a form of specification called an object dependency diagram.[1] The paradigm is not about how to create the specification.

Although much has been written on object-oriented design, SEI project members could find no examples of object-oriented implementations relevant to flight simulators. Examples were required for two reasons. First, object-orientation was new to both of the contractors on the ASVP. A methodology which leads to a specification of objects is useful only if developers know how to implement what is specified. Second, managers were skeptical about the benefits of object-oriented design. Examples were needed to determine whether benefits outweigh costs.

The intent of our work was to produce examples of object-oriented systems. It was not our intent to determine whether object-oriented design was best for flight simulators.[2]

[1] See Chapter 4 and Figure 4-1 for an example of an object dependency diagram.

[2] See Section 2.1 for some historical motivation.

1.3. Characteristics of the Application Domain

The paradigm was developed for a specific application domain, namely flight simulators and training devices. This section puts the paradigm in context by briefly describing the relevant features of the application domain.

The objective of a flight simulator is to reproduce on the ground the behavior of an aircraft in flight. Simulators are used to

- train aircrew,
- train maintainers of aircraft, and
- aid designers of aircraft.

A training simulator consists of a mock-up of stations for the aircrew being trained. The mock-up contains the controls available to manipulate the aircraft and systems for cuing the operator to the aircraft's response to his actions. Cues include gauges, video, sound, and motion.

The training mission is set by an instructor at an Instructor Operator Station (IOS). Some of the factors set by the instructor are longitude, latitude, altitude, and atmospheric conditions. They also affect the behavior of the simulator by introducing aircraft malfunctions.

The ASVP focused on software that models the behavior of major systems affecting an aircraft's flight: the airframe, the engines, the electrical system, the fuel system, the hydraulic system, and others.

Traditionally, this software is put under the control of an executive which periodically updates systems. Flight simulators are not event-driven. Interaction between systems in the real aircraft are continuous. Simulators model those interactions in discrete time.

Time constraints are normally tighter than memory constraints. Multiple processors are used to distribute processing and to link the software to hardware in the aircrew training station. Trends are such that multi-processor architectures are becoming more prevalent in the domain.

Flight simulators are long-lived and frequently modified. The two major causes of modification are modifications to the aircraft itself and changes in the training missions. Typical of the latter is the simulation of new malfunctions.

Flight simulators are based on math models provided by the manufacturer of the aircraft components in the actual aircraft. The ultimate test of the simulator is the way it feels to aircrew experienced with the aircraft being simulated. The process of tuning the feel of the simulator is called aircrew tuning.

Flight simulators provide natural opportunities for reusing software. First, different

aircraft have the same kinds of components, e.g., engines, fuel systems, electrical systems, etc. Sometimes a particular instance of a kind of component, a Pratt and Whitney engine for example, is used on a variety of aircraft. Second, the three classes of simulators—training, maintenance, and engineering—model the same components to varying degrees of fidelity. Third, a simulator is made up of systems that can be viewed identically at some level of abstraction.

1.4. Reader's Guide

This report contains the work completed to date, presents the paradigm, and discusses the advantages of the paradigm. It is meant to stand on its own merits. The model we have developed solves a specific set of problems. We do not claim it to be the only model for solving these problems. The paradigm uses many of the characteristic software engineering concepts, but the report is not intended to be a report on software engineering theory.[3]

The next chapter discusses our approach to developing the paradigm and how we assessed the fit of our solution to the problem at hand.

Chapter 3
: introduces the conceptual elements of the paradigm and provides an overview of the software structure implied by the paradigm.

Chapter 4
: presents a detailed view of the elements of the paradigm. The elements are presented bottom-up using an Engine system as an example. Each section on a particular element ends with a discussion of the benefits of the implementation chosen for the paradigm. The final section of Chapter 4 summarizes the benefits of the paradigm.

Chapter 5
: discusses the role of a paradigm in the development process.

Chapter 6
: discusses issues which we have thought about during the development but have not had time to fully address.

Chapter 7
: is a very brief presentation of a simulator Electrical system.

Appendix A
: describes a modified form of the notation expounded on by Grady Booch in his book on software engineering with Ada [1] and his book on reusable software components with Ada [2]. The notation is used in the diagrams in Chapter 3.

Appendix B
: contains an object manager template. The use of reusable code templates is discussed in Chapter 5.

Appendix C
: presents a version of the Engine system code complete through the package specifications. The intent is to demonstrate the software architecture defined by the object paradigm discussed in Chapter 4.

---

[3] If the audience perceives that this report would be useful within a tutorial on software engineering, we invite such a use of the report.

2. Approach

2.1. History

The project team began the search for a paradigm after reviewing an implementation of an electrical system done by one of the contractors on the ASVP. The implementation was more data-oriented than object-oriented. The implementation was a definite improvement over the original FORTRAN implementation. However, the team did not consider the implementation to be exemplary.

The project team decided to spend what it thought would be no more than a month or two developing an example of a pure object-oriented design of an electrical system. A circuit diagram was used to identify the objects and the relationships among the objects. The behavior of the objects, e.g., circuit breakers, relays, and batteries, and of circuits in general, was well understood.

Material available to us on object-oriented design did not adequately address connections among objects or updating systems of objects in discrete time.

The project team implemented an object-oriented electrical system which came close to satisfying the goals described below. At that time one of the contractors on the ASVP asked the project team to sketch out an object-oriented implementation of an engine. The team observed that the object-oriented implementation of an engine and of an electrical system were identical at some level of abstraction.

The project team decided to capture the similarities in a paradigm for object-oriented systems. The paradigm was to dictate how an object-oriented specification would be implemented in software and how the update of systems would be controlled. The drive to generalize uncovered flaws in our designs of both the engine system and the electrical system.

The project team did not develop the paradigm methodically. We were not interested in testing design methods. Our goal was to produce a paradigm for object-oriented systems. We did not want to limit our search space to architectures produced by known methods.

2.2. Design Goals

The project team began with two basic goals. One was to eliminate nested implementations of objects. The other was to simplify dependencies among objects.

Nested objects result from decompositional approaches that purport to help the designer discover which objects are needed to implement a system. For example, the designer begins with the notion of an engine as a black box. All interfaces to the engine appear at the surface of the black box. Now, suppose the vibration of an engine compressor needs to be metered. The designer decides to decompose the engine into other objects, one of which is a compressor. Access to the vibration level of the compressor passes through two levels: the engine level and the compressor level. Further, decomposition might lead to modeling each stage of the compressor as an object, thus adding a third layer to the nested object. Finally, black box implementations require knowledge of the entire black box, even when only one state or aspect of the black box is used.

Nested, hierarchical objects do have advantages. First, it should be possible to update a composite object, such as an engine, as if it were a black box. Second, it should be possible to reuse an object, such as an engine, as a separate entity.

Figure 2-1 shows a dependency between objects A and B. In this example, B provides A with something.[4] Thus the state of A depends on the state of B.[5] One common solution is to have the implementation of object A *with* object B. When A is updated, A reads the relevant state of B. This solution does not work if B and A are on separate processors. Even if A and B are on the same processor, the dependencies for devices as complex as flight simulators are complicated themselves. Also, it is never clear which object should define the dependent data type.

Another common solution is to have object B call object A and report its state. This solution introduces a new problem without solving the problem mentioned above. If the flow between B and A is continuous, then it is unnatural for object B to model discrete time by controlling the rate at which A is updated. Further, if B and A are part of a closed feedback loop, the update cycles indefinitely.

The major problems with the solutions discussed above involve the fact that objects in flight simulators interact through real-world entities, such as wires and pipes. The real-world connections are typically not modeled in software. Instead they are subsumed by procedure calls embodied in one of the objects.

---

[4]The same diagrammatic notation is used throughout this report. The dependent object is at the tail of the arrow. It depends on something from the object at the head of the arrow.

[5]In Ada, an object which depends on another, separately compiled object, uses the *with* clause to gain visibility of the dependent object. The object is said to *with* the dependent object.

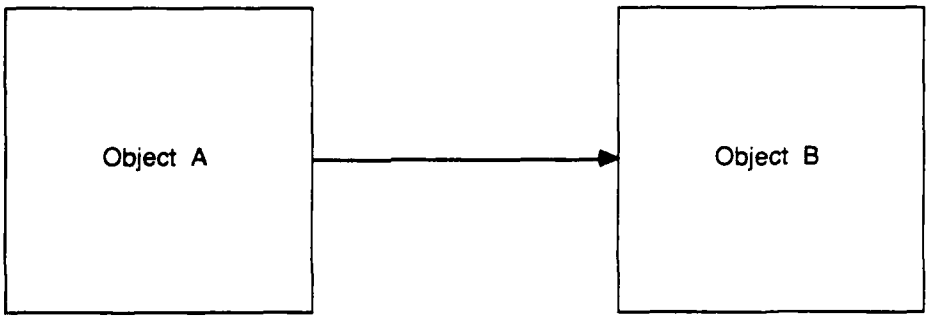


Figure 2-1: Object Dependency Example

2.3. Evolution of the Paradigm

Designers talk about the fit of a design to its context, the problem space. The criteria for assessing the fit of solutions to complex problems often can be determined only in response to a proposed solution and cannot be determined before solutions are generated. Such was the case for the paradigm.

Our team began with intuitive feelings about the standard goals of software engineering; goals such as modularity, ease of enhancement, and reuse. The paradigm passed through four or five iterations within the team. Each iteration left a legacy of criteria for assessing the fit of the solution for the paradigm.

For example, the model for object managers[6] and the means for connecting objects surfaced in the first version of the paradigm. The objects stood alone, and were not dependent on Ada types declared elsewhere. This enhanced the reusability of the object managers and facilitated independent development. The means for connecting objects had an intuitive analog in the real-world. Pipes and wires, connecting objects in the world, are as real as the objects themselves and should not be subsumed in software by the implementations of the objects.

In addition, the number of concepts was minimized. Those objects which had no analog in the physical world were removed.

The chapters which follow discuss the advantages of the paradigm. We did not set out to obtain these advantages. The advantages revealed themselves as the work progressed. An advantage which revealed itself in one iteration was retained as a criterion for evaluating the fit of subsequent iterations.

---

[6]Object managers are introduced in Chapter 4.

3. Concepts Used by the Paradigm

This chapter provides a brief description of some of the concepts introduced with the paradigm and a high level overview of the software architecture defined within the paradigm. The concepts are further elaborated in Chapter 4.

The paradigm described in this report begins with the notion of an *executive*. An *executive* controls the update of a set of systems compiled together running on a single processor. The paradigm assumes that there will be more than one set of systems and that multiprocessing will be involved.

Communication between executives is handled by an abstraction called a *buffer*. A *buffer* is some means of sharing data among separately compiled software.[7] The paradigm makes no assumption about how the operating system transfers data or how executives on separate processors are invoked.

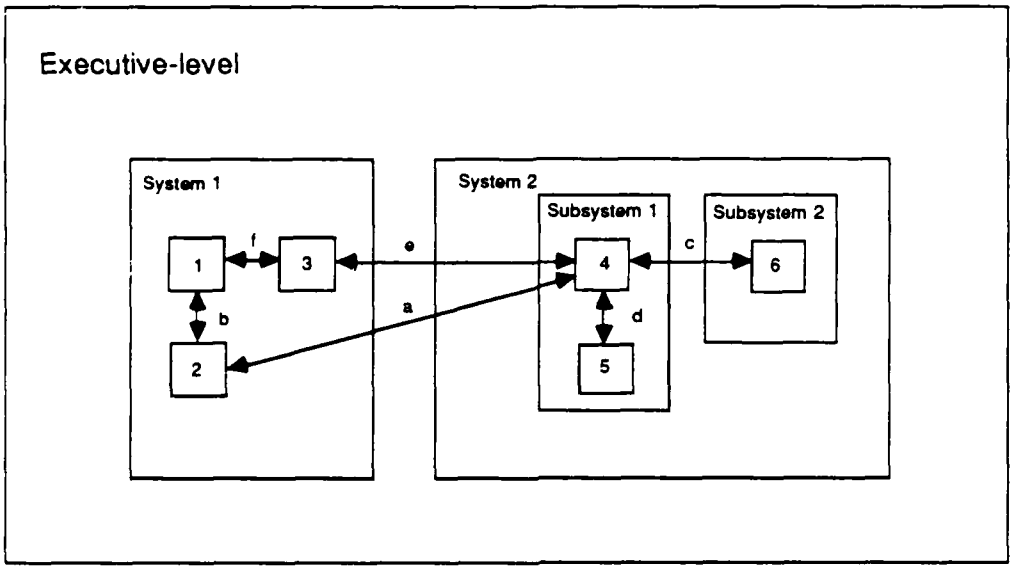
The fundamental units of the paradigm are *objects* and *connections*. *Objects* map to real-world entities. An *object* is implemented as a math model that maps the environmental effects on the object to the object's outputs, given the attributes of the object and its operational state. The implementation isolates individual effects. Also, an object is not aware of its connections to other objects.

A *connection* models real-world conduits and is the mechanism for transferring sate information between objects. Processing a connection involves reading the state of some objects on the connection and broadcasting to others.

At all levels, updates are accomplished by processing the appropriate connections. The three levels discussed in the paradigm are *subsystem*, *system*, and *executive*. A *subsystem* is an aggregation of objects and the connections among those objects. A *system* is a set of subsystems and the connections from objects in any subsystem in the set to objects in any other subsystem in the set. If a system has only one subsystem, then the system and the subsystem are identical. An *executive* is a set of systems and all connections that cross

[7] In our observations of flight simulators, a *buffer* is a record data structure used in the communication between processors.

system boundaries. Figure 3-1 shows views of an executive, two systems, and several subsystems and objects.



Executive is : System 1, System 2, and connections **a** and **e**
System 1 is : Objects 1, 2, and 3, and connections **b** and **f**
System 2 is : Subsystem 1, Subsystem 2, and connection **c**
Subsystem 1 is : Objects 4 and 5, and connection **d**
Subsystem 2 is : Object 6

Figure 3-1: Software Architecture Example

3.1. Overview of the Software Architecture

3.1.1. The Executive Level

Figure 3-2 shows the executive-level software architecture[8]. In this case, we assume an executive for Flight_Systems. The body of Flight_Systems contains a tabular schedule of subsystems to update. The names of the subsystems are declared in the package Flight_Subsystems_Names, the sole purpose of which is to enumerate the names.

Each system is represented by a package called **<system_name>_Updater**.[9] The specification of an Updater package exports a single procedure which is called by Flight_Systems to update a subsystem of the system. A single parameter tells the Updater which subsystem is to be updated.

---

[8]See Appendix A for a description of the notation used in Figures 3-2, 3-3, and 3-4

[9]The use of "<...>" within subprogram names, type names, or text refers to a general case of the item. For example, **<system_name>_Updater**, is a general form representing all instances of the package name, e.g., **Engine_Updater**, **System_Power_Updater**, etc. See Chapter 5 for a more detailed discussion and examples of the use of "<...>".

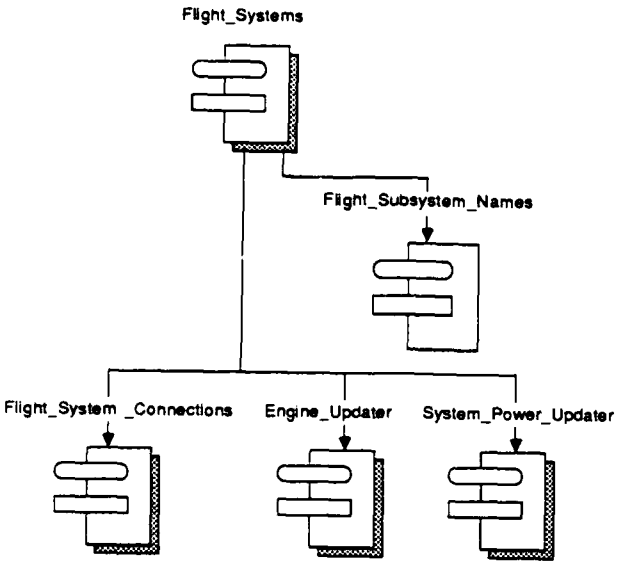


Figure 3-2: Executive Level Architecture

The connections belonging to the executive-level are managed by an **<executive_name>_Connections** package, in this case, Flight_System_Connections. The architecture of the connection package is shown in Figure 3-3.

The body of the connection package is a series of separate procedures, one for each system under the control of the executive. The separate procedures for systems with more than one subsystem take a subsystem name as an argument.

Each procedure updates system objects connected to objects outside the system. As discussed in the next chapter, objects are implemented as private types; pointers to the objects are stored in a data structure contained in a package, **<system_name>_Aggregate**.

3.1.2. System Level

Figure 3-4 shows the architecture of a system, using engines as an example. Objects in a system are created and named by the **<system_name>_Aggregate** package. Objects are managed by **<object_name>_Object_Manager** (OM) packages.

Systems with more than one subsystem use the names of its subsystems to differentiate among identical objects and similar sets of connections. Details on this aspect of the architecture are presented in the next chapter.

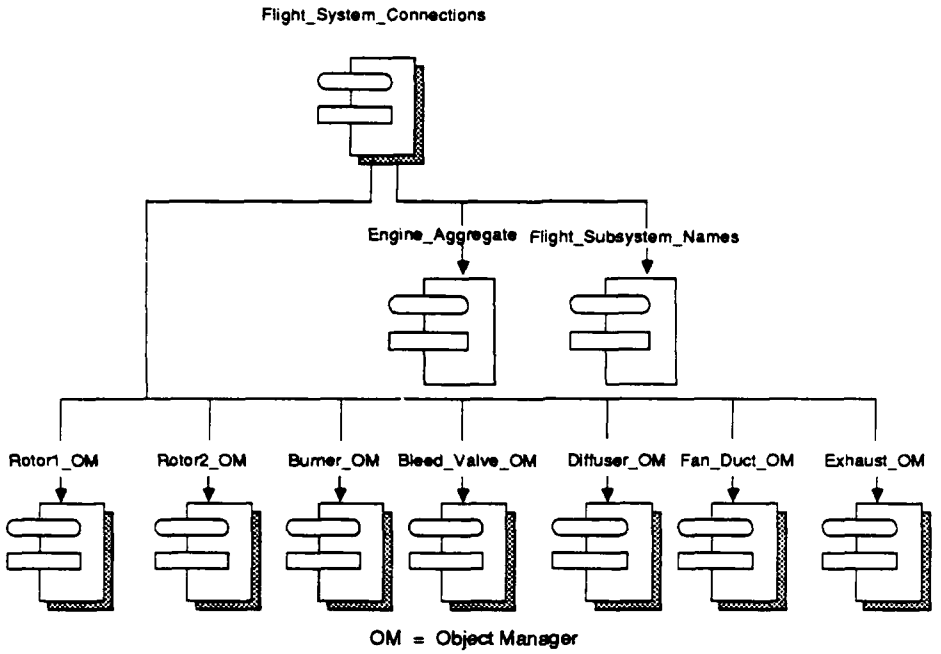


Figure 3-3: Connection Manager Architecture

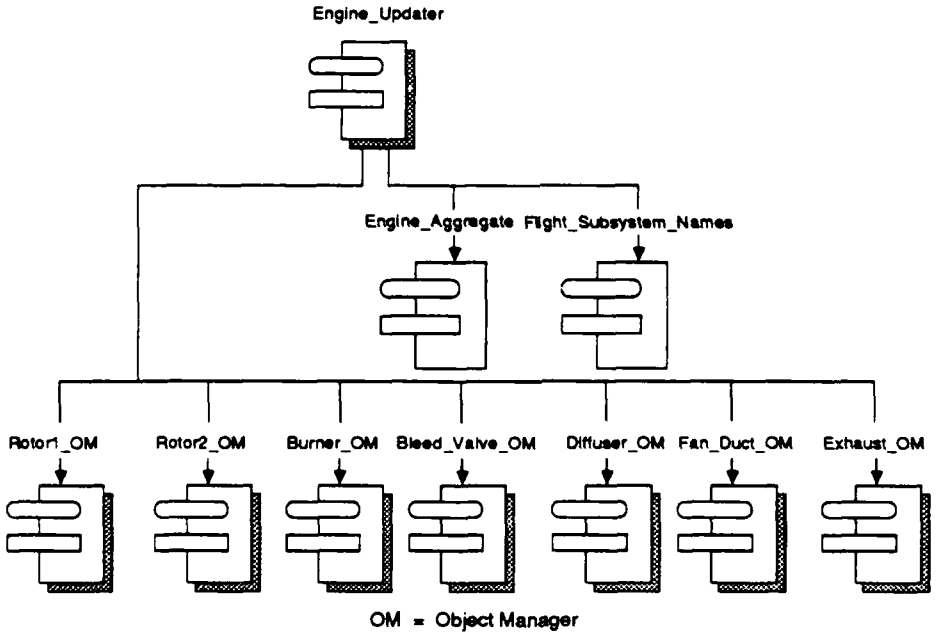


Figure 3-4: System Level Architecture

4. Paradigm Description

The first example to illustrate the paradigm is a turbofan engine. Engines, in flight training simulators, interact with a variety of other systems on the aircraft, including the fuel system, the oil system, the starter, the electrical system, and the hydraulic system. The engines also provide bleed air for cabin pressure and air conditioning.

The next section in this chapter will describe the engine components and the interaction of the engine with the rest of the aircraft systems. The following sections will describe the paradigm using the engine model as an example.

4.1. Engine Parts Description

The engine object dependency diagram in Figure 4-1 will be referred to throughout the rest of this chapter. The diagram represents the objects which comprise a generic turbofan engine and the engine's relationship with the outside environment. The process for identifying the objects is not an issue for this report. The choice of objects may not be ideal, but for the purposes of the discussion in this report, this set of objects is acceptable. For more information on turbofan engines, see [3].

The engine is the area within the large rectangle. The rounded rectangles external to the engine represent other systems in the aircraft, e.g., electrical system, fuel system, etc., or in the aircraft's environment, e.g., atmospheric and environmental conditions.

The square boxes within the rectangle represent the engine objects. The objects are:

- Diffuser
- Rotor1
- Fan Duct
- Rotor2
- Burner
- Bleed Valve
- Exhaust.

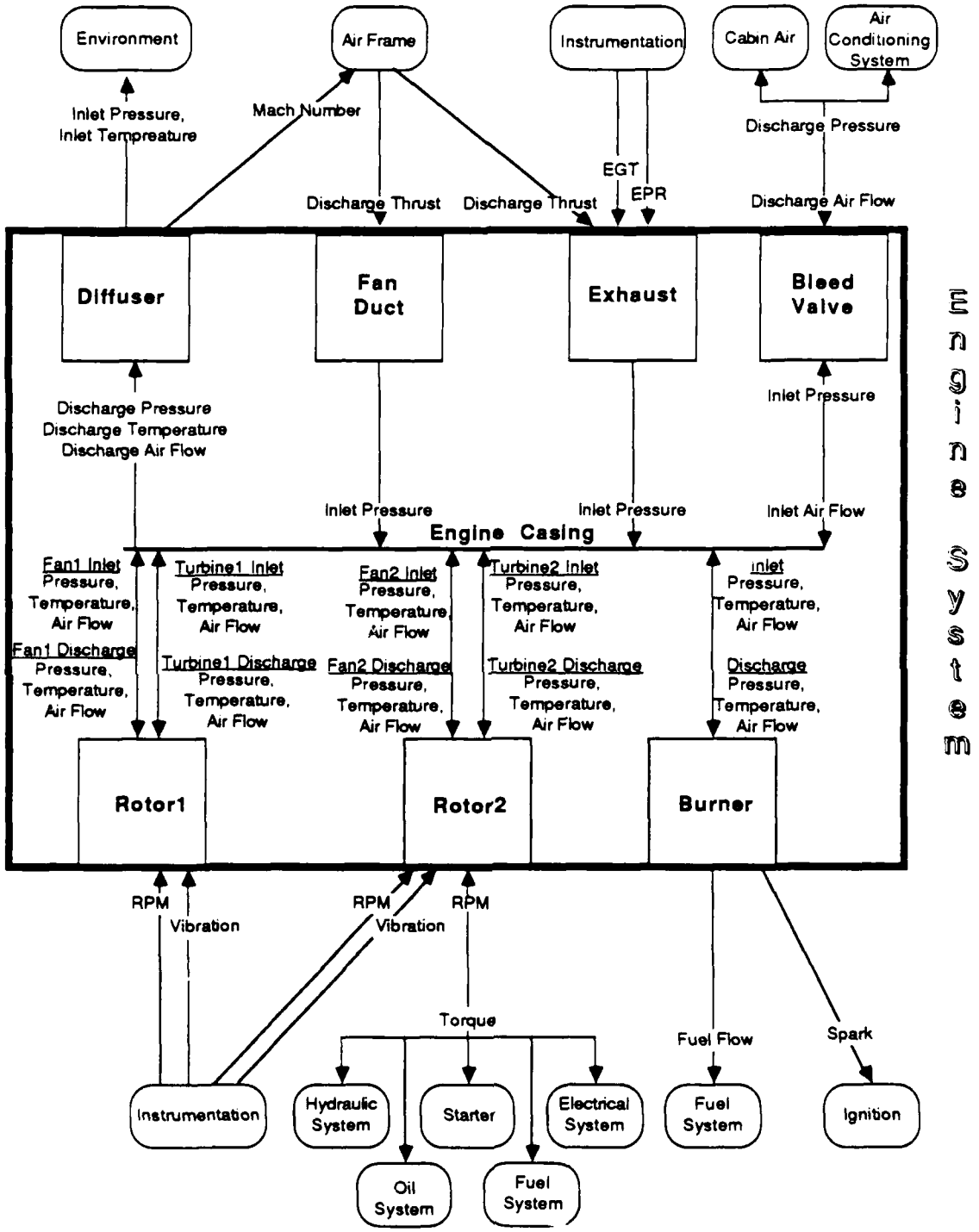


Figure 4-1: Turbofan Engine Dependency Diagram

The arrows represent dependencies among objects. A single-headed arrow points in the direction of the dependency, e.g., the **Diffuser** is dependent on the **Air Frame** for *mach number*, and the **Instrumentation** and the **Air Frame** are dependent on the **Exhaust** for other state information. A double-headed arrow represents dependencies in both directions, i.e., it is equivalent to two single-headed arrows. For example, the **Air Conditioning** is dependent on the **Bleed Valve** for a value of air flow, and the **Bleed Valve** is dependent on the **Air Conditioning** and the **Cabin Air** for a measure of the air pressure that they require. The arrows are labeled with the state information which is needed between the objects and the external systems.

The heavy line, labelled **Engine Casing**, is intended to represent the internal connection between the engine objects within the engine system. It is the connection through which the air flows as the air passes through the engine. Each object has some dependency on the air flow, as it passes through the connection, denoted by the arrows into the connection. Thus the **Rotor1** is dependent on the **Engine Casing** for its *Fan1 Inlet* air pressure, temperature, and flow. Each object also makes its outputs available to the **Engine Casing** for use by other engine objects, e.g., the **Rotor1** makes its *Fan1 Discharge* air pressure, temperature, and flow available to the **Engine Casing**.

Each engine object in the engine diagram interacts with its external environment as defined by the diagram. No other dependencies on the outside world should be necessary except for those shown in the diagram. The diagram serves as a specification for the engine system interfaces. Given such a diagram and the paradigm description that follows, the design of the engine system is complete.

Thus, an engine system is made up of the objects and connections between them inside the large rectangle. An aircraft simulator, for a multi-engine aircraft, would have multiple engine systems. Each would be handled identically, but would have different connections to the outside world.

4.2. Object Abstraction

The identification and extraction of objects from the problem space is not an issue here. This section describes an object abstraction assuming the objects are identified. The engine diagram in Figure 4-1 will serve as an example.

Objects correspond to real world entities. Objects generalize behavior, i.e., they know nothing about their environment and they are identical in each of the engines in a multi-engine system. They only differ in how they are connected to their environment. The objects, however, have no knowledge of these connections. The connections are described in Section 4.3. Finally, objects' internal states are consistent with the latest known external effects at all times.

4.2.1. Object Managers

Each object is represented by an object manager. There is a single object manager for all instances of the object.[10] Referring to the engine diagram, Figure 4-1, there will be an object manager for each of the objects in an engine:

- Diffuser
- Rotor1
- Fan Duct
- Rotor2
- Burner
- Bleed Valve
- Exhaust.

The object manager defines the attributes of the object. The attributes are invariant characteristics defined at elaboration, e.g., an ampere rating of a circuit breaker.

The object manager allows the object's environmental effects to be placed on the object. The environmental effects are external object states which are required by the object to determine its state. The environmental effects are placed on an object individually by connecting procedures. The procedures defined for these operations are described in Section 4.2.3.

The object manager implements the math model for the object. The math model is implementation dependent.

The object manager defines the operational state of the object. The operational state refers to those characteristics which may change with time, e.g., the frictional state of a rotor, malfunctions, or aging effects on various components.

The object manager defines the outputs available from the object. The outputs are generated by the math model, using the environmental effects placed on the object and any additional constraints imposed by the attributes and the operational state of the object. The math model may be invoked when environmental effects are placed on the object or when outputs are read from the object. This is an implementation level decision left to the system designer; it is not defined by the paradigm.

The object manager defines an interface to the operations available on an object. The operations allow the placing of environmental effects, updating the operational state, and reading the outputs of the object.

The actual instances of the object are stored in object aggregates which are discussed in Section 4.4.1. An aggregate allows named access to the objects; no procedure call is required to retrieve the object.

---

[10]The term *manager* is used because all access to each object is administered through the interface defined by the object manager.

Finally, the object manager is independent of the rest of the system. The only compilation dependencies are on global types.

4.2.2. Object Manager Structure

The representation of the object in an object manager is declared as a private type in the package specification. Figure 4-2 is a partial package specification containing typical type definitions found in an object manager.[11] Use of a private type allows external access to the object while hiding the details of the object's implementation. In addition, the package specification must define all the types used to describe the object's attributes, the operational state, and the placeholders for environmental effects.[12] For the **Burner** Object Manager in Figure 4-2, a type definition for Spark is provided. In the private part of the package specification, the object's private type is declared as an access pointer to a data type which will be the actual representation of the object. The data type is an incomplete type, the details of which are delayed until the package body.[13]

The object's data representation, defined in the package body, must allow for storage of environmental effects and reading of the object's outputs. A typical implementation is a record with components for each of the object's attributes, operational state variables, and placeholders for the environmental effects.. Each attribute component must have a default value and each operational state variable should have an initial state value.

4.2.3. Object Manager Operations

There are three types of operations within each object manager. There is also a standard naming convention for these operations. One side effect of the naming convention is that all object managers begin to look very similar. The similarity can be exploited to create an object manager template, see Chapter 5, which can be used to generate new object managers.

The first type of operation is used to create new instances of the object. This operation is a function, named **New_<object>**[14], which returns an instance of the private type, **<object>**. For example, in Figure 4-2, the function provided by the **Burner** object manager is called **New_Burner**; it returns an instance of the private type, **Burner**. This private type is a pointer to a new instance of the data type representing the object. In addition, values for

---

[11]Package **Standard_Engineering_Types**, *with*ed at the beginning of Package **Burner_Object_Manager** in Figure 4-2, contains several global definitions for typical simulator types. The package is shown in Appendix Section C.2.

[12]The attributes and operational state variables must be visible to the aggregate which instantiates the objects and to the system and executive level connections which operate on these objects. See Sections 4.4.1, 4.3, 4.4, and 4.5 for descriptions of aggregates, connections, systems, and executives.

[13]See Appendix Section C.4 for the complete Package Specification for the **Burner** object. Appendix C provides an implementation of the Engine system through the Ada specifications.

[14]The use of "<...>" within subprogram names, type names, or text refers to a general case of the item. For example, **New_<object>**, is the general form representing all instances of the *New* function, e.g., **New_Burner**, **New_Rotor1**, **New_Exhaust**, etc. See Chapter 5 for a more detailed discussion and examples of the use of "<...>".

```
with Standard_Engineering_Types;

package Burner_Object_Manager is

  type Burner is private ;     -- a Burner is an abstraction of a
                               -- Burner within an Engine.

  type Spark                   is (None, Low, High);

  function New_Burner return Burner;

  procedure Give_Inlet_Air_To(
    A_Burner              : in Burner;
    Given_Inlet_Pressure  : in Standard_Engineering_Types.Pressure;
    Given_Inlet_Temperature  : in Standard_Engineering_Types.Temperature;
    Given_Inlet_Air_Flow  : in Standard_Engineering_Types.Air_Flow
  );

  procedure Get_Discharge_Air_From(
    A_Burner              : in  Burner;
    Returning_Discharge_Pressure  : out Standard_Engineering_Types.Pressure;
    Returning_Discharge_Temperature: out Standard_Engineering_Types.Temperature;
    Returning_Discharge_Air_Flow  : out Standard_Engineering_Types.Air_Flow
  );

  procedure Give_Fuel_Flow_To(
    A_Burner              : in Burner;
    Given_Fuel_Flow       : in Standard_Engineering_Types.Fuel_Flow
  );

  procedure Give_Spark_To(
    A_Burner              : in Burner;
    Given_Spark           : in Spark
  );

private
  type Burner_Representation;          -- incomplete type, defined in
                               -- package body
  type Burner is access Burner_Representation;
                               -- pointer to a Burner representation

end Burner_Object_Manager;
```

Figure 4-2: Burner Object Manager

components of the data type, which need their default values changed or their initial values defined, may be set by the **New_<object>** function. Typically, this function is called at elaboration, i.e., during system initialization. The return value, a pointer which is the "ID" of the new object, is stored and used to access the object in later operations. See Section 4.4.1 for more discussion on this point.

The second type of operation is used to write external effects, i.e., environmental effects and operational state changes, on an object.. The naming convention for this operation is **Give_<outside_effects>_To**. The operation takes the object private type and either external environment values or new operational state values as arguments. In Figure 4-2, the procedure **Give_Inlet_Air_To** is an example of this type of operation.

The characteristics of the **Give_<outside_effects>_To** procedure are as follows:

- report external environmental effects to the object. The stored values of the environmental effects will be used the next time the object's outputs are calculated. These updates are typically under the control of a cyclic executive and are placed on the object one or more times each cycle.
- report a change in the operational state to the object. The stored values of the operational state variables will be used the next time the object's outputs are calculated. These changes are typically asynchronous events triggered by the instructor at the IOS.
- the environmental effects and operational state variables are "saved" with the object in the private data structure.
- the environmental values stored with the object are consistent with the external effects at all times.

Ideally, the math model isolates the individual effects of the environmental effects. Calculation of the object's outputs can be postponed until the object's internal state is read.

The interfaces defined by the **Give_<outside_effects>_To** operations can be read directly off the object diagram, Figure 4-1. There will be one procedure per dependency arrow. For example, in Figure 4-2, procedure **Give_Inlet_Air_To**, for the Burner Object Manager, takes the pressure, temperature, and air flow as arguments.

The third type of operation is used to read an object's outputs. The outputs are calculated by the math model using the environmental effects placed on the object and any additional constraints imposed by the attributes and the operational state of the object. The math model may be invoked when external effects are placed on the object or when outputs are read from the object. The naming convention for this operation is **Get_<object_output>_From**. The operation takes the object private type as an argument and returns the object's outputs. In Figure 4-2, the procedure **Get_Discharge_Air_From** is an example of this type of operation.

The characteristics of the **Get_<object_output>_From** operation are as follows:

- the response reflects the current state of the object. The state is dependent on the environmental effects previously placed on the object, the object's attributes, and the object's operational state. The outputs are read from the private data structure or calculated from the values stored in the data structure.
- the output state of the object is consistent with the external environmental effects at all times.
- each operation is specific to the object and the output of the object that it reports. This operation is the only way to access the object's output.

The interfaces defined by the **Get_<object_output>_From** operations can be read directly off the object diagram, Figure 4-1. There should be one procedure per dependency arrow. For example, in Figure 4-2, procedure **Get_Discharge_Air_From**, for the Burner Object Manager, returns the pressure, temperature, and air flow.

The output state of an object, determined from its environmental effects, attributes, and operational state, may be calculated either when new external information is written to

the object (and then the output state should be stored with the object), by the **Give_<outside_effects>_To** procedure, or when outputs are read from the object, by the **Get_<object_output>_From** operation. In the first case, each time an external effect is deposited, a new output state should be calculated and stored so that the correct output state can be returned on subsequent read operations. Since each external effect is independent of all others, the object's output state will be consistent at all times. In the second case, an object's output state is not stored, but calculated each time the outputs are read. The decision as to which implementation to use is up to the implementor of the system. That level of detail is not specified in the paradigm.

4.2.4. Advantages of the Object Abstraction

The implementation of objects as described in this chapter follows the standard model for object-oriented abstraction. The object managers embody the state of the object, and changes in the object's environment are placed on the object procedurally. The major difference is the removal of connections from the objects (connections are described in Section 4.3). This decision supports separate development of objects since there is no dependency on any modules other than global types. In addition, spaghetti compilation dependencies are prevented. Finally, reuse is supported, since typing differences between objects is not an issue.[15]

Another advantage of the object manager is to focus the addition of detail at one place. For example, if there is loss of efficiency in the movement of air through the **Burner**, the loss can be modeled in the object manager for the **Burner**. Also, malfunctions in components can be simulated in the objects. The introduction, handling, and reporting of a malfunction should be introduced at the object manager level.

4.3. Connection Abstraction

Objects are represented by the implementation scheme described in Section 4.2. At that point one has a pool of isolated objects.

This section describes connections, i.e. the mechanism for transferring state information between objects.

4.3.1. Overview of Connections

Software connections model real-world conduits.

The connection between the engine objects in Figure 4-1 is represented by the heavy line labeled **Engine Casing**. The arrows in the diagram represent dependencies. The arrowhead points in the direction of the dependency. A double-headed arrow represents dependency in both directions.

---

[15]One of the roles of connections is to convert types when necessary, see Section 4.3.

Connections are also used to capture the passage of time. The software clock can be viewed as another system, external to engines, which makes the current time (or elapsed time) available via a connection.

Connections provide a means to transfer information between buffers and software objects. The buffers may be a linkage buffer between the software and the simulator hardware, an Instructor Operator Station (IOS) buffer between the software and the IOS station, or buffers between processors in a multi-processor configuration. In all these cases, the connection handles the transfer of environmental effects or operational state information from the buffer to the software objects and the transfer of object state from the software objects to the buffer. For example, software lights in the electrical system can be turned on and off as a result of external environmental effects or operational state changes. These effects must be transferred to the simulator cockpit and affect a change in the hardware lights. Lights can also be turned on and off in the simulator cockpit by the students. These effects must be transferred to the software and change the operational state of the software lights. The linkage buffer between the cockpit and the software is used and connections handle the information flow.

Finally, the updating of a system is accomplished by moving information along connections.

4.3.2. Procedural Abstraction

Objects are oblivious to their environment. An object manager stores environmental effects and operational state information and provides access to the object's outputs.

The operations defined with the object allow for writing information to the object and reading information from the object. See Section 4.2.3 for more discussion on the object operations.

The connections between objects are captured procedurally, using these operations. All connections between objects within systems and between systems are modeled the same way.

The connecting procedures exist outside the object managers, but have visibility into the object managers.

The connecting procedures need to perform three steps:

- obtain the needed information directly from an object
- convert the information if necessary
- put the information directly onto another object.

Each step is discussed in more detail in the following sections.[16]

4.3.2.1. Get Needed Information

The initial step is to obtain the external information which must be placed on an object. The provider of the information is defined within an object diagram at the head of each arrow, as in the Engine diagram, Figure 4-1. The provider will be either an external system, e.g., the Fuel system, or another object within the Engine system. If the provider is from an external system, the procedure modeling the connection must have access into the objects of each system. Thus the procedure needs to exist at the next higher level of abstraction, i.e., within the enclosing executive. Within the executive, local variables may exist to allow for temporary storage of the information, as in Figure 4-3. The current value of *spark*, from the Ignition system object manager, is obtained with a call to **Get_Spark_From** and stored in the local variable **Some_Spark**. Thus, although the paradigm does not advocate careless typing, it recognizes that perfect type matches will not always be possible.

If the provider is from another object within the Engine system, then the enclosing scope of the objects, i.e., the Engine system itself, handles the connection.

4.3.2.2. Convert Information

The connecting procedures encapsulate type conversions. Each object manager maintains the state of the object in the units which make sense to that object. The connecting procedures handle the type conversions which are necessary between the object managers. In Figure 4-3,[17] the intermediate value, obtained during the get information step above, is converted to the proper enumerated type, as understood by the Burner object manager, by the function **Spark_Conversion**.

There are two reasons for managing type conversions within the connection procedure. First, the object managers are then free from inter-object type dependencies. The object managers become stand-alone, with no dependencies other than on global data types. Thus, the object managers become reusable units. Separate development of the object managers is also supported. The second reason is that each object manager has a different need. There is no reason to expect that the **Burner** object manager would have a need to know how the **Ignition** object manager maintains the *spark* state. For example, the *spark* from the **Ignition** system may be in volts while the Burner maintains the value as an enumerated type (see Figure 4-3).

---

[16]So far, the discussion has focused on the simple case of two objects per connection. For a connection with multiple objects, e.g., the connection between the **Rotor2** and the five external systems in Figure 4-1, the steps above expand to include each object:

- obtain the needed information directly from all objects
- process the collected information and convert if necessary
- put the information directly onto all objects.

[17]The notation used in Figure 4-3, **Engines(An_Engine).The_Burner**, is part of the Engine Aggregate nomenclature discussed in Section 4.4.1.

```
procedure Process_Engine_Connections_To (
        A_Subsystem: in Flight_Subsystem_Names.Name_Of_A_Flight_Subsystem) is
--
-- A local variable is defined to store the value spark when it is read from
-- the ignition system. This is a convention, described in the SEI Ada
-- Coding Guidelines, to restrict the spread of embedded function calls, i.e.
-- function calls as parameters within other function calls.
--
    Some_Spark : Ignition.Spark;

    function Spark_Conversion (In_Spark : in Ignition_Object_Manager.Spark)
        return Burner_Object_Manager.Spark is
    --|*********************************************************************
    --| Description:
    --|    This function performs a type conversion.  It converts
    --|    the spark from the Ignition to a spark that the
    --|    Burner_Object_Manager can accept.  This is done
    --|    as an example of how the type conversions can be used to
    --|    connect objects which either communicate through a
    --|    valve/regulator, or need different grains of coarseness of
    --|       the information.
    --|    In this case we are assuming that the Ingition system
    --|    needs finer information about the spark than the Burner.
    --|
    --| Parameter Description:
    --|    In_Spark is the spark that the Ignition supplies.
    --|    return Spark is the spark returned for the Burner
    --|*********************************************************************
    begin
        case In_Spark is
            when 0..2  => RETURN Burner_Object_Manager.None;
            when 3..9  => RETURN Burner_Object_Manager.Low;
            when 10..20 => RETURN Burner_Object_Manager.High;
        end case;
    end Spark_Conversion;

begin
--  .
--  .
--  .
-- get Spark from the Ignition and feed it to the Engine Burner.
--
    Some_Spark :=
        Ignition.Get_Spark_From (This_Ignition(Given_Engine_Name));

    Burner_Object_Manager.Give_Spark_To (
        A_Burner => Engines(An_Engine).The_Burner,
        Given_Spark => Spark_Conversion(Some_Spark));
--  .
--  .
--  .
-- and so on
--
end Process_Engine_Connections_To;
```

Figure 4-3: Spark Conversion Routine

Also, within the connecting procedure modeling of flow, resistance, or friction between objects is possible. For example, constriction within pipes or the presence of valves in a connecting line might alter the flow. Since the connecting procedures are modeling the flow

in the line, the variation in the flow along the line can also be modelled. The conversion that needs to take place is a change in the value of the flow rate.

4.3.2.3. Put Converted Information

The final step is to place the external environmental information on the object being updated. The information must be in the proper type to match the dependent object. Once again, a picture, like that in Figure 4-1, defines the destination for the environmental information. The procedure **Give_Spark_To** (Figure 4-3) is an example of a put information operation.

4.3.3. Advantages of Connections

The implementation of connections in connecting procedures, as described in this chapter, provides a consistent and natural interface to the objects.

The connections insulate the objects from compilation dependencies. Objects, subsystems, and systems become stand-alone. Each can be developed independently. Connecting procedures provide a firewall; changes in implementation to objects on one side of a connection do not affect the implementation of objects on the other side.

Connections facilitate independent development and reuse. In particular, connecting procedures provide a systematic way to handle typing mismatches. The type conversions between objects are easily managed since the connecting procedures have visibility into the objects.

Connecting procedures provide a consistent means of updating systems and objects. Thus, connecting procedures provide a means for specifying control flow. No extraneous concepts or operations are required. The notion of connecting procedures handles all types of interactions between objects.

The connecting procedures provide a locus of control since all connections at an abstraction level are handled in one place.

Finally, the modeling of malfunctions is facilitated, i.e., they can be defined easily and implemented just like any other connection between two objects.

4.4. Subsystems and Systems

To this point we have defined objects and the connections between them. This section discusses a method for grouping the objects and connections together into a logical scope.

A *subsystem* is an aggregate of objects and the connections between the objects. The objects are accessible by name outside the subsystem, as discussed below. The connections among the objects are hidden at the subsystem level, i.e., no higher level has any knowledge of the subsystem connections.

A subsystem update involves the processing of the subsystem level connections, as described in Section 4.3. No access is required to objects outside the subsystem.

A *system* is a set of related subsystems. One example is the Engine system which consists of a group of identical but isolated subsystems, one for each engine. The same objects are used in each subsystem; the connections to the outside world are different; and there are no connections between the subsystems. Another example is a simulator Electrical system, which consists of related and tightly coupled subsystems. The same objects are used in each subsystem; the same kind of connections are used throughout; and each subsystem depends on the others.

If there is only one subsystem, then the system and the subsystem are the same. A system can see only the objects belonging to its subsystems. The connections among objects in the subsystems that cross subsystem boundaries are the responsibility of the system and are called system level connections.

All the subsystems in a system can be updated at the same time or individually. The update is initiated by an executive call to the system. If the system has more than one subsystem, then a parameter is used to specify the subsystem. The system level connections to the subsystem are processed, then the subsystem is updated, as described above. No access is required to objects outside the system.

4.4.1. Subsystem Aggregates

A real world system usually consists of collections of objects. An aggregate creates and names a collection of objects. An aggregate is a data structure indexed by the name of the object. Objects are accessed by name. A procedure call is not required to obtain a "pointer" to the object being accessed.

4.4.1.1. Building an Aggregate

As was described in Section 4.1, an engine is a collection of objects, including the diffuser, rotors, a burner, and so forth. Each object is managed by its own object manager. An engine record can be constructed as a grouping of these objects (see the **Engine_Representation** in Figure 4-4).

```
package Engine_Aggregate is

  type Engine_Representation is      -- Defines an engine representation
  record                             -- as consisting of an:
   The_Diffuser  :  Diffuser;
   The_Rotor1    :  Rotor1;
   The_Fan_Duct      :  Fan_Duct;
   The_Rotor2    :  Rotor2;
   The_Bleed_Valve  :  Bleed_Valve;
   The_Burner    :  Burner;
   The_Exhaust   :  Exhaust;
  end record ;

end Engine_Aggregate;
```

Figure 4-4: Engine Representation Example

For an aircraft as a whole there may be several engines. Using a constant array, an aggregate of the engines can be created which stores references to **Engine_Representations**, one for each engine on the aircraft (see Figure 4-5). The constant array, **Engines**, is created at elaboration time. Each object is instantiated by a call to the function **New_<object>**, described in Section 4.2.3, with all initial conditions set by default. The pointer to the private type returned by the function is stored with the name of the object. Thus, reference to the object can be done by name. The aggregate data structure is visible so no procedure call is required to retrieve an object. The array is indexed by the enumerated engine names **Engine_1..Engine_4**. The engine names are defined in a global type package that defines all the subsystem names.

The constant array, **Engines**, is defined in a package specification to allow access to the Engine system by an external system which *with*s the package specification and the appropriate object managers. The aggregate and object managers are used by the connecting procedures, discussed in Section 4.3.3, to reference the necessary objects. All references to objects are done through the aggregates. An object in an engine is referenced as:

```
Engines(Engine_Name).The_<object>
```

For example, the **Diffuser** of Engine 1 is referenced as:

```
Engines(Engine_1).The_Diffuser
```

and the **Rotor1** of Engine 3 is referenced as:

```
Engines(Engine_3).The_Rotor1
```

The code fragment, in Figure 4-6, shows how to reference an engine object using the aggregate.[18] The *Discharge Air* is read from the **Diffuser** object using the reference, **Engine_Aggregate.Engines(Given_Engine_Name).The_Diffuser** and written to the **Rotor1** object using the reference, **Engine_Aggregate.Engines (Given_Engine_Name).The_Rotor1**.

4.4.2. Updating

The existence of subsystems allows the processing of the enclosed objects to be done as a unit. The process of updating a subsystem occurs in two steps (shown for an executive in Figure 4-8):

- process the external connections and
- perform the subsystem update.

The operations are done atomically for each subsystem. This means that when it is time to update a subsystem, all work necessary to complete both steps of the update is finished before the process is begun on another subsystem.

Processing the external connections involves calling the connecting procedures, as described in Section 4.3. The external effects, i.e., effects from objects external to the subsystem, are processed by the connecting procedures at the enclosing levels.

---

[18]The **Given_Engine_Name** used to reference the objects will be passed as a parameter to the connecting procedure performing the update.

```
package Engine_Aggregate is

  -- Define an object which holds all 4 engines in the system and
  -- initialize them (i.e. all their parts).
  Engines: constant array (Engine_1..Engine_4) of Engine_Representation :=
   (Engine_1  => (
     The_Diffuser   =>   New_Diffuser,
     The_Rotor1     =>   New_Rotor1,
     The_Fan_Duct   =>   New_Fan_Duct,
     The_Rotor2     =>   New_Rotor2,
     The_Bleed_Valve => New_Bleed_Valve,
     The_Burner       =>   New_Burner,
     The_Exhaust      =>   New_Exhaust
     ),
    Engine_2  => (
     The_Diffuser   =>   New_Diffuser,
     The_Rotor1     =>   New_Rotor1,
     The_Fan_Duct   =>   New_Fan_Duct,
     The_Rotor2     =>   New_Rotor2,
     The_Bleed_Valve => New_Bleed_Valve,
     The_Burner       =>   New_Burner,
     The_Exhaust      =>   New_Exhaust
     ),
    Engine_3  => (
     The_Diffuser   =>   New_Diffuser,
     The_Rotor1     =>   New_Rotor1,
     The_Fan_Duct   =>   New_Fan_Duct,
     The_Rotor2     =>   New_Rotor2,
     The_Bleed_Valve => New_Bleed_Valve,
     The_Burner       =>   New_Burner,
     The_Exhaust      =>   New_Exhaust
     ),
    Engine_4  => (
     The_Diffuser   =>   New_Diffuser,
     The_Rotor1     =>   New_Rotor1,
     The_Fan_Duct   =>   New_Fan_Duct,
     The_Rotor2     =>   New_Rotor2,
     The_Bleed_Valve => New_Bleed_Valve,
     The_Burner       =>   New_Burner,
     The_Exhaust      =>   New_Exhaust
     )
   );

end Engine_Aggregate;
```

Figure 4-5: Engine Aggregate Example

Once all the external effects have been placed on the subsystem, then a subsystem update is performed through a single procedure call, **Update_<subsystem>**, to the subsystem.

For the Engine system example, all the subsystems, i.e., each engine on the aircraft, are independent of each other. There is no information which has to pass between the subsystems. The only external connections which need to be processed are those from systems outside the Engine system, e.g., the Fuel system. These connections are handled at the enclosing executive level. Then a subsystem is updated, via the procedure **Update_Engine**, with the subsystem as a parameter, see Figure 4-6. Performing the subsystem update in-

```
package Engine_Updater is

procedure Update_Engine(Given_Engine_Name: in Name_Of_A_Flight_Subsystem) is

   Diffuser_Discharge_Pressure     : Standard_Engineering_Types.Pressure;
   Diffuser_Discharge_Temperature: Standard_Engineering_Types.Temperature;
   Diffuser_Discharge_Air_Flow     : Standard_Engineering_Types.Air_Flow;

  begin
  --
  -- Model the connection characterized by the dependence of the Rotor1
  -- on the Diffuser for Pneumatic_Energy.
  --
  -- NOTE, no type conversion is necessary because both types are based
  -- on Standard_Engineering_Types Package definitions.
  --
   Diffuser_Object_Manager.Get_Discharge_Air_From(
     A_Diffuser => Engine_Aggregate.Engines(Given_Engine_Name).The_Diffuser,
     Returning_Discharge_Pressure  =>     Diffuser_Discharge_Pressure,
     Returning_Discharge_Temperature =>  Diffuser_Discharge_Temperature,
     Returning_Discharge_Air_Flow   =>   Diffuser_Discharge_Air_Flow
   );
   Rotor1_Object_Manager.Give_Fan1_Inlet_Air_To(
     A_Rotor1   => Engine_Aggregate.Engines(Given_Engine_Name).The_Rotor1,
     Given_Fan1_Inlet_Pressure      =>    Diffuser_Discharge_Pressure,
     Given_Fan1_Inlet_Temperature   =>    Diffuser_Discharge_Temperature,
     Given_Fan1_Inlet_Air_Flow      =>    Diffuser_Discharge_Air_Flow
   );
  --
  --     .
  --     .
  --     .
  --

end Engine_Updater;
```

Figure 4-6: Reference to an Engine Object using the Aggregate

volves processing the connections at the subsystem level. The update procedure is dependent on the Engine Aggregate and the object managers. A fragment of the **Update_Engine** procedure is in Figure 4-6. The connection representing the dependency of the **Rotor1** on the **Diffuser** for air flow, temperature, and pressure is shown.

4.4.3. Advantages of Subsystems and Systems

The implementation of systems, as described in this chapter, encapsulates subsystems, objects, and connections within a logical scope. A system needs to access only its aggregated objects, the global types used by the objects, and the internal connections.

This separation of concerns allows for several things:

- Minimum compilation dependencies. Subsystems and systems become standalone. Connecting procedures provide a firewall; changes in implementation to objects in a subsystem on one side of a connection do not affect the implementation of objects in another subsystem on the other side.
- Separate development of components and reuse. Systems and subsystems are self-contained. The only dependencies are on global types and object managers.

- a potentially easy disbursement within a multi-processor environment (more on this in Section 6.1).

4.5. Executives

An executive is a system consisting only of other systems. For example, the Flight Systems Executive surrounds the Engine system, the Electrical system, the Fuel system, etc. The executive controls the updating of all the systems within its scope. The executive handles all connections between its systems, e.g., those between the Engine system and the Fuel system. In a multi-processing environment, in this model, there would be one executive level per processor. The executive would have buffers for communication between the processors. However, the synchronization among the processors would happen outside the executive.

```
with Flight_Systems_Connections;
with Flight_Subsystem_Names; use Flight_Subsystem_Names;
with Global_Types;

 package Flight_Systems is

   type Active_In_Frame is array (Name_Of_A_Flight_Subsystem)
       of Boolean;

   Its_Time_To_Do : constant array (Global_Types.Execution_Sequence) of
              Active_In_Frame:=
     (Frame_1_Modules_Are_Executed => (Engine_1 => (True),
                        others => (False)),
      Frame_2_Modules_Are_Executed => (Ac_Power => (True),
                          others => (False)),
      Frame_3_Modules_Are_Executed => (Engine_2 => (True),
                        others => (False)),
      Frame_4_Modules_Are_Executed => (Dc_Power => (True),
                          others => (False)),
      Frame_5_Modules_Are_Executed => (Engine_3 => (True),
                        others => (False)),
      Frame_6_Modules_Are_Executed => (others => (False)),
      Frame_7_Modules_Are_Executed => (Engine_4 => (True),
                        others => (False)),
      Frame_8_Modules_Are_Executed => (others => (False))
      );


end Flight_Systems;
```

Figure 4-7: Executive Activity Table Example

4.5.1. Implementation of an Executive

All the subsystems within the executive's systems are known to the executive, as are all the object's in those subsystems. The executive has an activity table, indexed by the subsystems, which defines an order for processing those subsystems. An implementation for use within a cyclic executive is shown in Figure 4-7. The constant array, *Its_Time_To_Do*, defines the frame in which each subsystem within the Engine system and the Electrical

system gets processed. The processing is actually initiated by the procedure shown in Figure 4-8.

```
with Global_Types;

with Flight_Systems_Connections;
with Flight_Subsystem_Names; use Flight_Subsystem_Names;

with Engine_Updater;
with System_Power_Updater;

 package Flight_Systems is

procedure Update_Flight_Systems (Frame: in Global_Types.Execution_Sequence) is
--|*********************************************************************
--| Description:
--|    flight systems executive.  Performs process connections and update
--|    as an atomic action for each subsystem.
--|
--| Parameter Description:
--|    frame is the current executing frame
--|
--| Notes:
--|    none
--|*********************************************************************

   begin
       for A_Subsystem in Name_Of_A_Flight_Subsystem loop
           if Its_Time_To_Do (Frame)(A_Subsystem) then
               case A_Subsystem is

                   when Dc_Power..Ac_Power =>
                       Flight_Systems_Connections.
                           Process_Power_Connections_To (A_Subsystem);
                       System_Power_Updater.
                           Update_System_Power(A_Subsystem);

                   when Engine_1..Engine_4 =>
                       Flight_Systems_Connections.
                           Process_Engine_Connections_To (A_Subsystem);
                       Engine_Updater.
                           Update_Engine (A_Subsystem);
               end case;
           end if;
       end loop;

end Update_Flight_Systems;

end Flight_Systems;
```

Figure 4-8: Flight Executive Example

The processing for a subsystem involves putting the outside effects on the subsystem and then telling the subsystem to update itself. These operations for the subsystems are done atomically. For example, in Figure 4-8, when it is time to update an engine subsystem, a call is made to **Flight_Systems_Connections.Process_Engine_Connections_To**. This procedure accesses the engine objects directly, using the engine aggregate, to write outside

effects onto the engine objects. Figure 4-9 shows a fragment of a connecting procedure from the executive level. The fragment reads the torque energy required by the **Integrated Drive Generator** object manager in the Electrical system.

Next, the procedure **Engine_Updater.Update_Engine** is called, for the same engine subsystem, to process the connections internal to that subsystem. When this operation is finished, the engine subsystem update is complete and the subsystem is consistent with all its external effects.

```
Integrated_Drive_Energy :=
    Integrated_Drive_Object_Manager.Get_Torque_From (
        An_Integrated_Drive =>
            Integrated_Drive_Generators(A_Subsystem)
    );
```

Figure 4-9: Executive Connection Procedure Example

```
Integrated_Drive_Energy :=
    Flight_Systems_Buffer.Get_Torque_From (
        A_Buffer_Location =>
            Flight_Buffer.Idg(A_Subsystem)
    );
```

Figure 4-10: Communicating with a Data Transfer Buffer

4.5.2. Advantages of Executives

The implementation of executives described in this chapter follows the same model of connections used at the system and subsystem levels. Additionally, the executive has scheduling information in the form of an activity table which defines an order for processing its systems. Using the activity table, tuning of the simulator system by balancing the subsystem processing across the frames of the cyclic executive is simplified.

Distributed processing can be handled easily by partitioning executives across the available processors. More discussion of this topic is in Section 6.1.

4.6. Advantages of the Architecture of the Paradigm

The two main design goals for the paradigm were to eliminate unnecessarily layered objects and to simplify dependencies among objects. Both goals have been met.

The structure of objects is flat. Connecting procedures—the means, within the paradigm, of accessing states of objects—at the executive level can access all objects in systems under the control of the executive. Objects are accessed by name through the data structures which aggregate subsystem objects. A procedure call is not required to obtain a "pointer" to the object being accessed. We assert that the solution is natural. A spark goes to a burner, not to an engine.

The abstraction of higher-level objects, such as engines, is captured in the notion of a system, i.e., a set of objects updated as an entity. The benefits of nested objects are retained, i.e., high-level objects can be updated and reused as a single entity. This abstraction coupled with the approach to processing connections facilitates multiprocessing. Placing a set of systems on a separate processor requires only creating an executive for the processor and making minor changes to the executive-level connections to the system. None of the system-level code changes.

The major difference between this paradigm and other object-oriented paradigms is the use of connecting procedures to propagate changes. Connecting procedures allow objects, subsystems, and systems to standalone. Each can be developed independently. Connecting procedures provide a firewall: Changes in implementation to objects on one side of a connection do not affect the implementation of objects on the other side.

Connecting procedures facilitate both independent development and reuse. In particular, connecting procedures provide a systematic way to handle typing mismatches. It is desirable, but not always possible, for two connected objects to use the same types to communicate. Similarly, connecting procedures are a convenient way to adjust the performance of flight simulators to the expectations of crew members.[19]

The software partitioning of connecting procedures simplifies compilation dependencies. All access to objects happens through connecting procedures. Thus, it is only the procedures managing connections to a subsystem that need to be recompiled if an object manager specification changes. Each of these is implemented as a separate procedure.

Connecting procedures provide a consistent means of updating systems and objects. Thus, connecting procedures provide a means for specifying control flow. No extraneous concepts or operations are required. The notion of connecting procedures handles all types of interactions between objects.

The paradigm produces software that is easy to modify. Typical modifications include adjusting the distribution of processing among the frames of a cyclic executive, adding malfunctions, adding or removing objects, and modeling wear and aging of components. Examples of some of the potential modifications are:

1. moving the update of a subsystem to a different frame requires a change only in the executive's schedule table
2. adjusting the air flow for one of the systems using air flow can be done in the connecting procedure without worrying about side-effects in the other systems
3. adding a malfunction to an engine component, the burner for example, requires only the following:

[19]For example, referring to Figure 4-1, consider the five-way connection passing *torque* and *rpm* between **Rotor2** and five external systems. The connection procedure provides an easy locus to modify the effects on one of the external systems without affecting the other four. Typical implementations must be very careful that changing the communication mechanism doesn't perturb the way all the systems react.

a. making the malfunction selectable at the Instructor Operator Station (IOS)

b. adding a connection from the IOS buffer to the burner

c. changing the model of the burner.

4. the major math models of the engine need not be disturbed by changes; adding a third compressor stage to the engine requires only creating the object in software and changing the model of the casing accordingly

5. modeling function in a rotor due to wear on a bearing requires adding the interface **Time_Has_Passed (Amount: Time)** to the object, making a small change to the private type, and reducing the efficiency of the rotor in proportion to its time in service

6. adding a malfunction to a connection, e.g., the line to the burner, requires creating an object to save the state of the line and a connection from that object to the IOS buffer.

5. Development Process

5.1. Role of the Paradigm

The development of systems using the paradigm is a design activity. The paradigm molds the designer's analysis of the requirements. The paradigm accommodates objects and connections. The result of the analysis of the requirements is a set of real-world objects and connections grouped into subsystems and systems. Once this choice is made, the paradigm dictates the implementation.

The paradigm can be viewed as a means of consistently specifying objects, connections, subsystems, systems, and executive-levels. The result is a consistent implementation. Maintainers do not need to learn the architecture of each system. If the paradigm is followed, all systems will look the same.

During acquisition, the architecture of each system does not need to be evaluated. The quality of the architecture that follows from the paradigm needs to be evaluated only once. Design reviews can focus on the analysis of requirements, the choice of objects and connections, and the subsystem and system groupings.

5.2. Templates and Reuse

The software architecture defined within the paradigm consists of levels of abstraction. Each level, e.g., object manager level, subsystem level, system level, and executive level, has defined software components: object managers, updater packages, aggregates, and connection packages.

Each of these components is similar across different systems. This similarity can be exploited to create reusable templates for each component.

The templates contain the general features of the component, with place-holders for the specific features. Appendix B contains a complete object manager template. The template uses the notation **<object>** as a place-holder for the name of the object. The notation **<attribute_x>** is used for expression of operational state variables and attributes. The object operations are expressed in similar terms (See Figure 5-1).

```
function New_<Object> (
    <Attribute_1> : in <Object>_<Attribute_1>;
    <Attribute_2> : in <Object>_<Attribute_2>
    ) return <Object>;
--|*****************************************************************
--| Description:
--|    creates a new <object> as a private type.
--|
--| Parameter Description:
--|    <attribute_1> ...
--|    <attribute_2> ...
--|    <object> is the access to the private data representaion.
--|
--|*****************************************************************


procedure Give_<State_1>_To (
    A_<Object>: in <Object>;
    A_<Object>_Side: in <Object>_Side_Names;
    <A_State_1>: in <State_Type_1>);
--|*****************************************************************
--| Description:
--|    places <state_type_1> on a specific side of a <object>.
--|
--| Parameter Description:
--|    a_<object> is the <object> being acted on.
--|        a_<object>_side is the side of the <object> to be updated.
--|    <state_type_1> is declared ...
--|*****************************************************************

function Get_<State_1>_From (
    A_<Object> : in <Object>;
    A_<Object>_Side : in <Object>_Side_Names
    ) return <State_Type_1>;
--|*****************************************************************
--| Description:
--|    Reads <state_type_1> available at a specific side of a <object>.
--|
--| Parameter Description:
--|    a_<object> is the <object> being acted on.
--|    a_<object>_side is the side being queried.
--|    <state_type_1> is declared ...
--|*****************************************************************
```

Figure 5-1: Object Manager Template Example

The templates are not intended to contain all the necessary details for generating a complete version of the code. They are intended as a starting point. The framework for each object manager, update package, connection package, and aggregate is similar. The details are different. Package bodies and subprogram bodies are provided within the templates. The implementor provides details within a template's framework. The resulting components will have a similar look and structure. This will aid readability, understanding, and maintenance.

5.2.1. Diagram Parsers

Several commercial tools have the capability of parsing diagrams and generating code templates to varying levels of detail. The detail is limited by the diagram notation.

The dependency diagram, Figure 4-1, is typical of a diagram for which a parser could be written. The parser could generate the templates discussed earlier. We view this as a natural extension of the paradigm toward a more automated solution.

5.3. Enhancements to Object/Connection Diagrams

The notation used on the object diagram, Figure 4-1, reflects the dependencies between objects and state information. It defines the connections necessary to construct the system.

Several extensions to the diagram notation can be envisaged. One would be to delineate the processing order of the connections. The heavy line, labelled **Engine Casing**, is intended to represent the internal connection between the engine objects within the engine system. It is the connection through which the air flows as the air passes through the engine. Each object has some dependency on the air flow, as it passes through the connection, denoted by the arrows into the connection. Nothing on the diagram denotes the order of connection processing. There may, however, be a specific order necessary to insure a consistent state of the Engine system.

Another extension would be to add pointers to algorithms. The algorithms, expressed in pseudocode, could be inserted in package bodies by the diagram parser.

6. Open Issues

6.1. Distributed Processing

One of the design goals of the paradigm was to facilitate spreading the work load over multiple processors. The description that follows encompasses our theories on what would be required to distribute the processing over several processors. We have not implemented or tested any of these ideas.

The paradigm begins with the notion of an executive. An executive controls the update of a set of systems compiled together and thus running on a single processor. The paradigm assumes that there will be more than one set of systems and that multiprocessing will be involved.

The abstraction of higher-level objects, such as engines, into systems allows a set of objects to be updated as an entity. This abstraction coupled with the paradigm's approach to processing connections facilitates multiprocessing. Placing a set of systems on a separate processor requires only creating an executive for the processor and making minor changes to the executive-level connections to the system.[20] None of the system-level code changes.[21]

Communication between executives is handled by an abstraction called a *buffer*. A *buffer* is some means of sharing data among separately compiled software.[22] The paradigm makes no assumption about how the operating system transfers data or how executives on separate processors are invoked. For example, assume that the Flight System Executive has been split so that some of its systems, e.g., the Electrical system and the Fuel system, are on a processor separate from the Engine system. The executive that handles the Engine system

[20]A typical minor change is demonstrated in Figures 4-9 and 4-10.

[21]There are many approaches to the solution of this problem. We do not intend to compare or delineate all possible solutions. One other solution would be to have the generation of connection dependencies handled by compiler pragmas. The effects would be the same, however. Our goal was to minimize perturbations to the connection procedures.

[22]In our observations of flight simulators, a *buffer* is a record data structure used in the communication between processors.

needs to communicate with a buffer to get the environmental effects from these other subsystems. Figure 4-10 shows how communication between the executive's connecting procedure and a buffer can be implemented.[23] The fragment reads the torque energy required by the **Integrated Drive Generator** object manager in the Electrical system from the buffer. This is one of the changes required to implement a system on distributed processors.

Another required change would be to load the buffer with the states of objects needed by systems on the other processor. All outputs required by systems on other processors must be written into the buffer. This step would take place after the update of the subsystem, as defined in Section 4.4.2.

One can imagine a development environment which automatically accommodates the distribution of systems across processors. The notations for the object/connection diagram could be extended to indicate which systems were to be grouped on a processor. The "address" of the object read by a connection procedure could be calculated at link time: The choices would be an object or a buffer surrogate.

6.2. Tuning

The construction of a system using the paradigm results in a product which is easy to read, understand, and maintain. The performance of the system, however, still must fit into the time constraints demanded by the application. The implementation described in the paradigm (and embodied in the templates) is intended to be a starting point for a usable system. We fully expect that adjustment of some of the concepts may be necessary. For example, Ada allows an implementor to *inline* certain procedures and functions. The overhead of a procedure call is saved. For many of the object manager operations, which are only a few lines long and tend to be called frequently during an update, *inlining* may provide a significant time savings.

Another useful technique is that of combining effects. For example, providing multiple parameters to a subprogram instead of making multiple subprogram calls. The implementation of the Engine system, described in Chapter 4, demonstrates this technique. Figure 4-6 provides an example which shows three effects in each subprogram call.

A second method for combining effects is to group like objects together. For example, in a simulator electrical system there are hundreds of circuit breakers. Each one has to be updated with respect to the hardware linkage buffer on each cycle. Also, at each level several breakers have to be updated through their connections to other systems. One solution is to create an object manager that handles groups of identical objects. A circuit breaker collection manager would contain subprograms for dealing with groups of breakers at a time. Thus, a single subprogram call operating on a group of objects replaces multiple calls each operating on individual objects.

---

[23]The figure contains the same example used in Section 4.5.1, Figure 4-9.

6.3. Reposition and Flight Freeze

Flight freeze and reposition are two of the software modes of an aircraft simulator.

In flight freeze mode the simulator software state is frozen, i.e., it stops changing with time. Communication with the simulator hardware must be maintained. Freeze may be initiated by the instructor at any time during a training exercise when communication with students is necessary.

The reposition mode is initiated by the instructor at the IOS when a particular training exercise is to be repeated. The communication between the simulator software and the hardware is maintained, and new values for flight data are loaded into the software. After a sufficient waiting period to allow the software to ramp to the new conditions, the simulator is restarted.

The paradigm considers time to be an outside effect on an object. Thus, it might be possible to implement flight freezes by controlling the time effects on objects. Similarly, a reposition would be accomplished by using reposition connecting procedures. In reposition mode, the executive level would connect systems to reposition buffers. A connecting procedure would read from the buffer instead of the object it reads from during normal run mode.

We have not implemented or tested these ideas. However, we are convinced that the paradigm does not complicate reposition and flight freeze.

7. Electrical System

An Electrical system in an aircraft provides electrical power to devices in other systems: Devices such as fuel pumps and valves in the Fuel system, hydraulic pumps in the Hydraulic system, and air conditioning in the Environmental Control system. The systems are able to function only if power is available. They, in turn, put their load, i.e., the amount of current they require, back onto the Electrical system. The load is transferred back to the generators, along the Electrical system buses, where a determination of possible overloading takes place.

A subset of the Electrical system has been completed and tested. The code with accompanying documentation is available on request from the authors. The code illustrates some concepts not illustrated by the Engine system example.

7.1. Additional Concepts

The Engine system, Appendix C, is complete through the package specifications. The subset of the Electrical system is fully functional and has been thoroughly tested.

Several performance issues arose during the implementation. There are several hundred circuit breakers in a typical electrical system. Each one has to be updated with respect to the hardware linkage buffer on each cycle. Also, at each level of the software several breakers have to updated through their connections to other systems. The subprogram calls in each object manager were *inlined* in order to reduce the overhead during update.

Grouping of like effects is also performed. Voltage and load conversion factor (lcf) are updated together. In addition, voltage, lcf, and current are grouped in a data structure which is used during all read operations from objects. Both steps result in fewer subprogram calls.

The concept of updating a system as a unit means, to us, that all aspects of the system update must be complete in the execution frame. The subset includes a tie bar, an electrical bus which connects several other buses. In order to insure that the update is complete within the frame, the tie bar is processed repeatedly in the frame. The number of times necessary depends on the number of other connections to the tie bar.

Other issues that arose during the complete implementation included decisions about writing effects to objects and reading outputs from objects. For some objects, like circuit breakers, the external effects are written and outputs are calculated during a read operation. For other objects, states are calculated when effects change.

The Electrical system consists of related and tightly coupled subsystems. The same objects are used in each subsystem, the same kind of connections are used throughout, and each subsystem depends on the others. Thus, unlike the Engine system, there are connections at the subsystem level. The Flight executive updates the connections between other systems and the Electrical system. The Electrical system then updates the connections between its subsystems. Finally, each subsystem updates its internal connections. The multilevel updating cries out for the creation of object managers for collections of objects. We have chosen not to implement these; they are left as an exercise for the reader.

The Electrical system object/connection diagrams look like circuit diagrams. Given a library of objects and a diagram parser, one could fully automate the production of code from a circuit diagram.

Appendix A: Software Architecture Notation

The notation used to describe software architecture is a modified form of the notation expounded on by Grady Booch in his book on software engineering with Ada [1] and his book on reusable software components with Ada [2]. The notation used is true to the intent of Booch's notation. The variations (i.e., extensions) are:

- we use reduced package, subprogram and task icons inside larger icons rather than the object (or blob) icon
- we use object dependency arrows more subtly, to distinguish different types of dependencies
- we have layered the diagrams, i.e., we show a diagram of top level dependencies and then expand the bodies of the figures to show the next layers of detail
- we do not show the internal details of any reusable subsystem, package, subprogram or task which is used.

One final note about the notation: The figures need not show all the fine-grained detail of a package or subprogram. When the code of a package (or subprogram) is compared to a figure associated with that package (or subprogram) there may be nested procedures or packages not shown on a particular picture, or it may depend on a package not explicitly shown in the figure. The guidelines for these cases are:

- utility packages or services are not shown (this includes things like text_io, reusable data structure packages, math libraries, etc.)
- the figures are meant to show the significant details at a particular level, not all the details
- the definition of "a significant detail" is solely at the discretion of the designer.

Based on these ideas, Figures A-1 thru A-4 explain the meaning of each of the icons available using this notation.

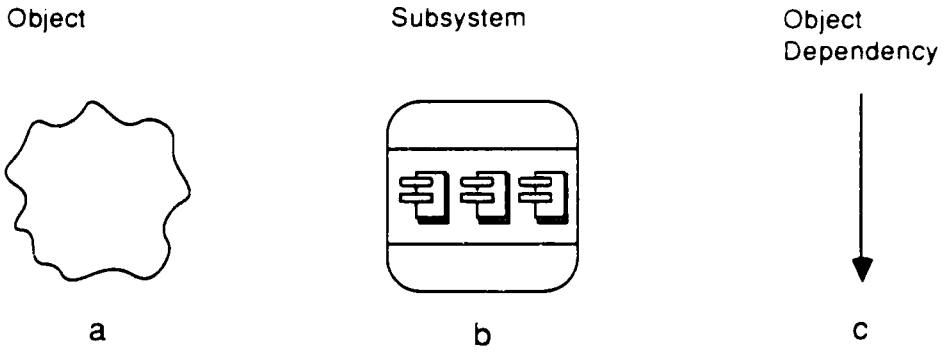


Figure A-1: Object, Subsystem and Dependency Notation

The object (or blob) icon, shown above in Figure A-1 (a), represents an identifiable segment of a system, about which we have no implementation information.

The subsystem icon, shown above in Figure A-1 (b), represents a major system component that has a clearly definable interface, yet, which is not representable as a single Ada package.

The object dependency symbol, shown above in Figure A-1 (c), indicates that the object at the origin of the arrow is dependent on the object at the head of the arrow. The origin of the arrow indicates where the dependency occurs. If the origin is in the white area of an icon (shown in subsequent figures), it indicates a specification dependency. If the origin is in the shaded area, it indicates a body dependency.

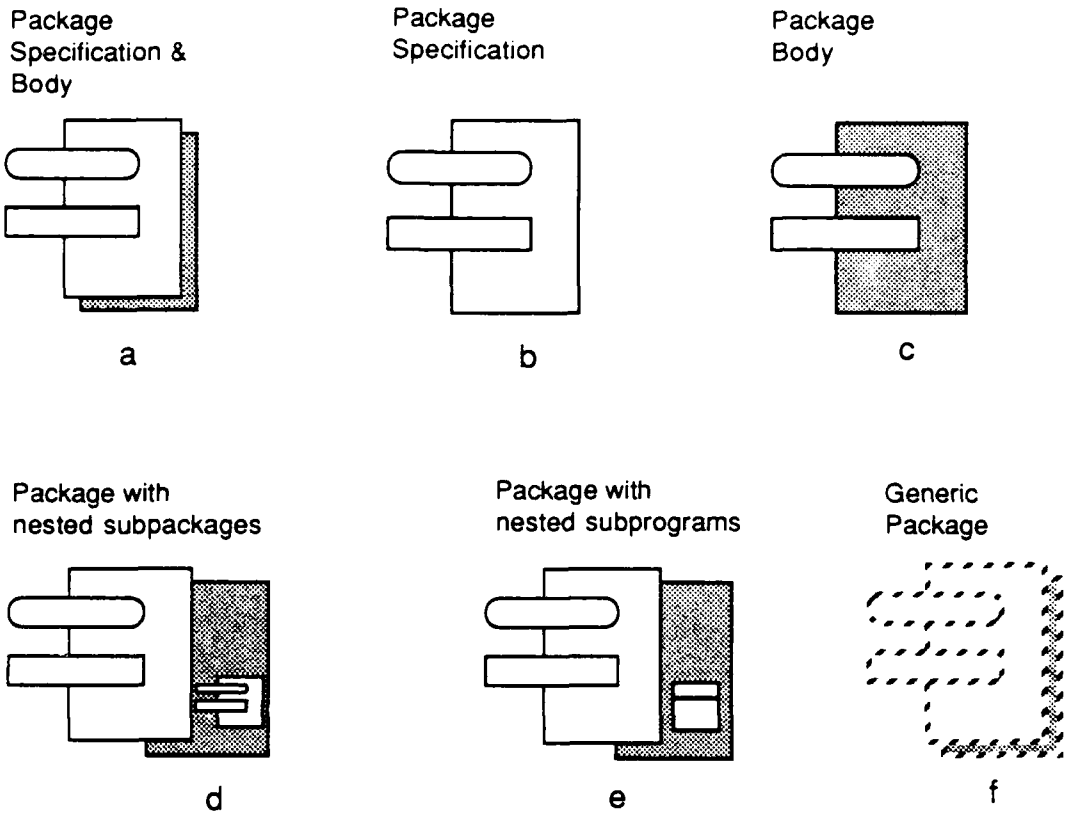


Figure A-2: Package Notation

The package specification and body icon, shown above in Figure A-2 (a), represents an Ada package specification, the white area, with an associated package body, the shaded area. This icon can be broken apart to show a package specification, Figure A-2 (b), or a package body, Figure A-2 (c).

Figures A-2 (d) and (e) are variations on the package icon which show greater detail. Figure A-2 (d) is used to represent packages which have nested subpackages within the body; if the small package icon were placed within the specification, it would indicate visible nested packages. Similarly, Figure A-2 (e) illustrates the notation used for separate subprograms within the body of a package.

Finally, Figure A-2 (f) illustrates the icon used for generic packages. Everything discussed above in regard to regular packages can also be applied to generic packages.

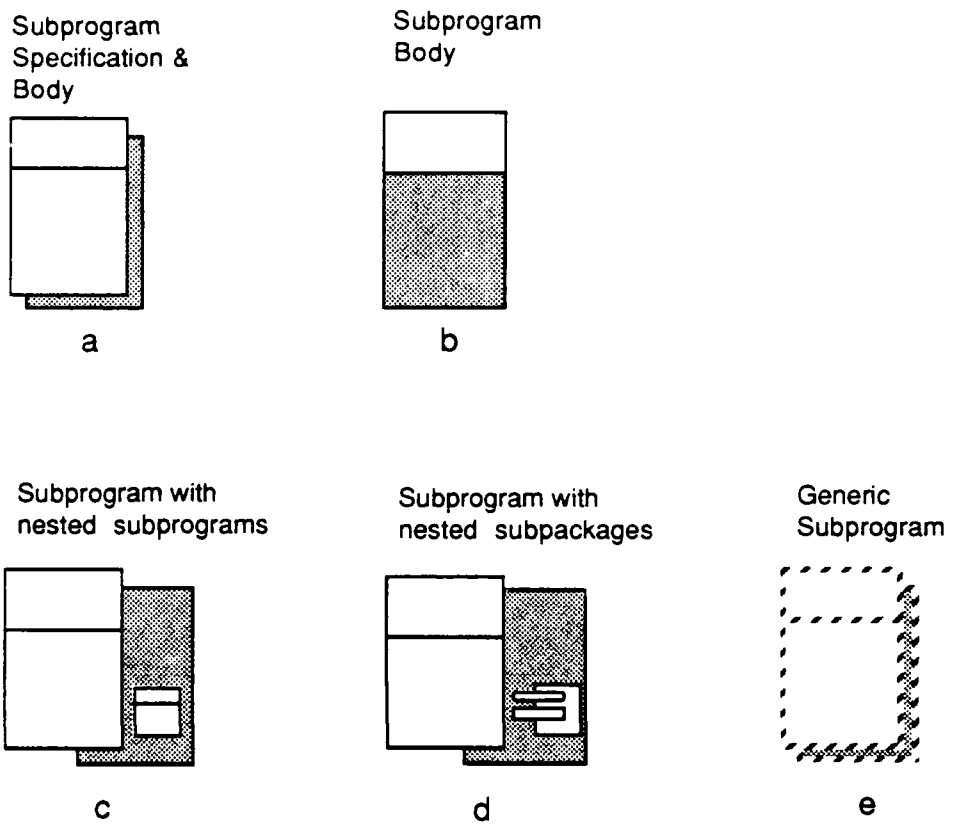


Figure A-3: Subprogram Notation

Much of what was discussed previously in regard to packages also applies to subprograms. The subprogram specification and body icon, shown above in Figure A-3 (a), represents an Ada subprogram specification, the white area, with an associated subprogram body, the shaded area. This icon can be broken apart to show a subprogram body, Figure A-3 (b).

Figures A-3 (c) and (d) are variations on the subprogram icon which show greater detail. Figure A-3 (c) is used to represent subprograms which have nested subprograms within the body. Similarly, Figure A-3 (d) illustrates the notation used for separate subpackages within the body of a subprogram.

Finally, Figure A-3 (f) illustrates the icon used for generic subprograms. Everything discussed above in regard to regular packages can also be applied to generic subprograms.

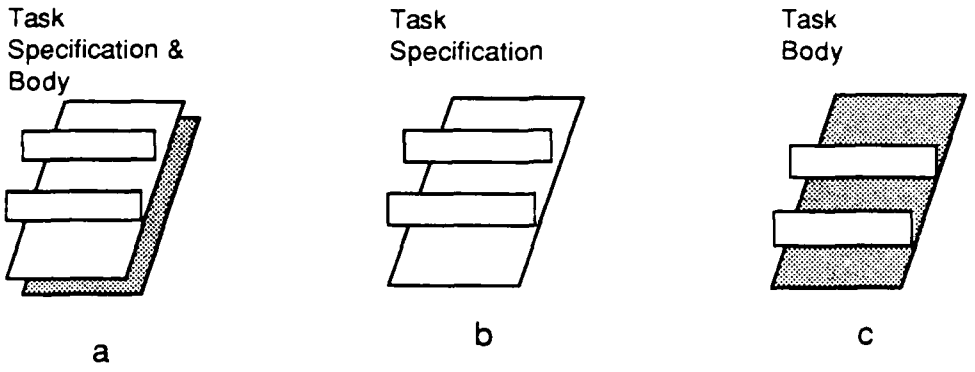


Figure A-4: Task Notation

Again, much of what was discussed previously in regard to packages and subprograms, applies to tasks. The task specification and body icon, shown above in Figure A-4 (a), represents an Ada task specification, the white area, with an associated task body, the shaded area. This icon can be broken apart to show a task specification, Figure A-2 (b), or a task body, Figure A-4 (c). Although they are not shown, nested package and subprograms are represented in exactly the same manner as shown in Figure A-2 for packages and subprograms.

Appendix B: Object Manager Template

```
--|*****************************************************************
--| Module Name:
--|    <object>_object_manager
--|
--| Module Type:
--|    Package Specification
--|
--| Module Purpose:
--|    implements the type manager for objects of type <object>.
--|-----------------------------------------------------------------
--| Module Description:
--|    <object> is implemented as a private data type created by new_<object>.
--|    Connections are connected to a side of an <object>; the sides of <object>
--|    are <sides>. Operations are available to read power units
--|    from and place power units on a specific Point to an <object>.
--|    The external states of <object> are <attribute_1>, <attribute_2>.
--|    These states are affected by actions other than the propagation of
--|    voltage and current within the subsystem of which a given
--|    <object> is a part. Operations are available to get and update
--|    these states.
--|
--| References:
--|    Design Documents:
--|
--|    User's Manual:
--|       none
--|
--|    Testing and Validation:
--|       none
--|
--| Notes:
--|    <object> is an element of an electrical circuit. Elements are
--|    connected to Connections. A connection reads power available
--|    from elements connected to it and propagates information to
--|    elements. The element
--|    retains information about conditions at all Points. The
--|    availability of power to a Connection depends on the state of
--|    the element and conditions on the opposite side of the element.
--|
--|-----------------------------------------------------------------
--| Modification History:
--|    ddMmmyy   author   Created
--|
--|-----------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    TBD
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense.
```

```
--      The views and conclusions contained in this document are
--      solely those of the author(s) and should not be interpreted as
--      representing official policies, either expressed or implied,
--      of the Software Engineering Institute, Carnegie Mellon
--      University, the U.S. Air Force, the Department of Defense,
--      or the U.S. Government."
--|*****************************************************************

with Electrical_Units; use Electrical_Units;

<package Object>_Object_Manager is

    type <Object> is private ;

    type <Object>_Side_Names is (<Sides>);

    type <Object>_<Attribute_1> is ...;

    type <Object>_<Attribute_2> is ...;


function New_<Object> (
    <Attribute_1> : in <Object>_<Attribute_1>;
    <Attribute_2> : in <Object>_<Attribute_2>
    ) return <Object>;
--|*****************************************************************
--| Description:
--|    creates a new <object> as a private type.
--|
--| Parameter Description:
--|    <attribute_1> ...
--|    <attribute_2> ...
--|    <object> is the access to the private data representaion.
--|
--|*****************************************************************


procedure Give_<State_1>_To (
    A_<Object>: in <Object>;
    A_<Object>_Side: in <Object>_Side_Names;
    <A_State_1>: in <State_Type_1>);
--|*****************************************************************
--| Description:
--|    places <state_type_1> on a specific side of a <object>.
--|
--| Parameter Description:
--|    a_<object> is the <object> being acted on.
--|        a_<object>_side is the side of the <object> to be updated.
--|    <state_type_1> is declared ...
--|*****************************************************************


procedure Give_<State_2>_To (
    A_<Object>: in <Object>;
    A_<Object>_Side: in <Object>_Side_Names;
    <A_State_2>: in <State_Type_2>);
--|*****************************************************************
--| Description:
--|    places <state_type_2> on a specific side of a <object>.
--|
--| Parameter Description:
--|    a_<object> is the <object> being acted on.
--|        a_<object>_side is the side of the <object> to be updated.
--|    <state_type_2> is declared ...
--|*****************************************************************
```

```
function Get_<State_1>_From (
    A_<Object> : in <Object>;
    A_<Object>_Side : in <Object>_Side_Names
    ) return <State_Type_1>;
--|*************************************************************
--| Description:
--|   Reads <state_type_1> available at a specific side of a <object>.
--|
--| Parameter Description:
--|   a_<object> is the <object> being acted on.
--|   a_<object>_side is the side being queried.
--|   <state_type_1> is declared ...
--|*************************************************************


function Get_<State_2>_From (
    A_<Object> : in <Object>;
    A_<Object>_Side : in <Object>_Side_Names
    ) return <State_Type_2>;
--|*************************************************************
--| Description:
--|   Reads <state_type_2> available at a specific side of a <object>.
--|
--| Parameter Description:
--|   a_<object> is the <object> being acted on.
--|   a_<object>_side is the side being queried.
--|   <state_type_2> is declared ...
--|*************************************************************


procedure Give_<Attribute_1>_To (
    A_<Object> : in <Object>;
    <Attribute_1> : in <Object>_<Attribute_1>);
--|*************************************************************
--| Description:
--|   Updates the state of <attribute_1> to correspond to current
--|   external conditions
--|
--| Parameter Description:
--|   A_<object> is the <object> to be updated
--|   <attribute_1> is the new <object>_<attribute_1>
--|
--|*************************************************************


function Get_<Attribute_1>_From (
    A_<Object> : in <Object>
    ) return <Object>_<Attribute_1>;
--|*************************************************************
--| Description:
--|   reads the state of <attribute_1> to correspond to current
--|   external conditions.
--|
--| Parameter Description:
--|   A_<object> is the <object> to be updated
--|   <attribute_1> is the current <attribute_1>
--|
--|*************************************************************


procedure Give_<Attribute_2>_To (
    A_<Object> : in <Object>;
    <Attribute_2> : in <Object>_<Attribute_2>);
--|*************************************************************
--| Description:
--|   Updates the state of <attribute_2> to correspond to current
```

```
--|    external conditions
--|
--| Parameter Description:
--|    A_<object> is the <object> to be updated
--|    <attribute_2> is the new <object>_<attribute_2>
--|
--|*************************************************************************

function Get_<Attribute_2>_From (
     A_<Object> : in <Object>
     ) return <Object>_<Attribute_2>;
--|*************************************************************************
--| Description:
--|    reads the state of <attribute_2> to correspond to current
--|    external conditions.
--|
--| Parameter Description:
--|    A_<object> is the <object> to be updated
--|    <attribute_2> is the current <attribute_2>
--|
--|*************************************************************************


private
     type <Object>_Representation;
     type <Object> is access <Object>_Representation;

end <Object>_Object_Manager;

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
pragma Page;
```

```
--|*****************************************************************
--| Module Name:
--|    <object>_Object_Manager
--|
--| Module Type:
--|    Package Body
--|
--|-----------------------------------------------------------------
--| Module Description:
--|    Reads and manipulates private data structures that represent
--|    a <object>.
--|
--| Notes:
--|
--|-----------------------------------------------------------------
--| Modification History:
--|        ddMmmyy   author      Created
--|
--|-----------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    TBD
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense.
--|    The views and conclusions contained in this document are
--|    solely those of the author(s) and should not be interpreted as
--|    representing official policies, either expressed or implied,
--|    of the Software Engineering Institute, Carnegie Mellon
--|    University, the U.S. Air Force, the Department of Defense,
--|    or the U.S. Government."
--|*****************************************************************

 <package Object>_Object_Manager is

    type Point_Representation is array (<Object>_Side_Names) of ...
    --
    -- representation of a <object>
    --
    type <Object>_Representation is
        record
            Points : Point_Representation;
            <Attribute_1> : <Object>_<Attribute_1>;
            <Attribute_2> : <Object>_<Attribute_2>;
    end record ;

pragma page;
```

```
function  Opposite_Side (This_Side : in  <Object>_Side_Names
     ) return  <Object>_Side_Names is
--|*************************************************************
--| Description:
--|        A function to find the opposite side of a Point.
--|        Requests for information about one side depend
--|        on the state of the <object> and information kept about
--|        the other side.
--|
--| Parameter Description:
--|        this_side is the side for which the opposite is sought.
--|        <object>_side_names is the opposite side.
--|
--| Notes:
--|    USED FOR CONTROL ELEMENTS SUCH AS CBs, RELAYs AND
--|    SWITCHES.
--|*************************************************************

     The_Side : <Object>_Side_Names := Side_1;

     begin
     --
     -- select opposite side based on what this side is.
     --
          if This_Side = Side_1 then
               The_Side := Side_2;
          end if;

          RETURN The_Side;

     end  Opposite_Side;

function  New_<Object> (
     <Attribute_1> : in  <Object>_<Attribute_1>;
     <Attribute_2> : in  <Object>_<Attribute_2>
     ) return  <Object> is
--|*************************************************************
--| Description:
--|    creates a new <object> and returns an access to it.
--|
--| Parameter Description:
--|    parameters are values for the attributes of <object>
--|    <object> returned is an access to the private type.
--|
--| Notes:
--|    uses the new operation to create record.  The
--|    temporary variable used to hold the access while
--|    the attribute values or set makes the code
--|    easier to read.
--|*************************************************************

     The_New_Object : <Object> := new  <Object>_Representation;

     begin
          The_New_Object.<Attribute_1> := <Attribute_1>;
          The_New_Object.<Attribute_2> := <Attribute_2>;

          RETURN The_New_Object;

     end  New_<Object>;


pragma page;
```

```
procedure Give_<State_1>_To (
    A_<Object>: in <Object>;
    A_<Object>_Side: in <Object>_Side_Names;
    <A_State_1>: in <State_Type_1>) is
--|*********************************************************************
--| Description:
--|   places <state_type_1> on a specific side of an <object>.
--|
--| Parameter Description:
--|   a_<object> is the <object> being acted on.
--|       a_<object>_side is the side to be updated.
--|   <state_type_1> is declared ...
--|*********************************************************************

    begin
        A_<Object>.Points (A_<Object>_Side).Xxx := <A_State_1>;

    end Give_<State_1>_To;


procedure Give_<State_2>_To (
    A_<Object>: in <Object>;
    A_<Object>_Side: in <Object>_Side_Names;
    <A_State_2>: in <State_Type_2>) is
--|*********************************************************************
--| Description:
--|   places <state_type_2> on a specific side of an <object>.
--|
--| Parameter Description:
--|   a_<object> is the <object> being acted on.
--|       a_<object>_side is the side to be updated
--|   <state_type_2> is declared ...
--|*********************************************************************

    begin
        A_<Object>.Points (A_<Object>_Side).Yyy := <A_State_2>;

    end Give_<State_2>_To;


pragma Page;
```

```
function Get_<State_1>_From (
    A_<Object> : in <Object>;
    A_<Object>_Side : in <Object>_Side_Names
    ) return <State_Type_1> is
--|*************************************************************
--| Description:
--|   reads <state_type_1> available at a specific side of an <object>.
--|
--| Parameter Description:
--|   a_<object> is the <object> being acted on.
--|   a_<object>_side is the side queried
--|   <state_type_1> is declared ...
--|*************************************************************

    The_<State_Type_1> : <State_Type_1>;

    begin
        if A_<Object>.<Attribute_1> = Xxxxx then
            The_<State_Type_1> := A_<Object>.Points (
                Opposite_Side (A_<Object>_Side));
        end if;

        RETURN The_<State_Type_1>;

    end Get_<State_Type_1>_From;


function Get_<State_2>_From (
    A_<Object> : in <Object>;
    A_<Object>_Side : in <Object>_Side_Names
    ) return <State_Type_2> is
--|*************************************************************
--| Description:
--|   reads <state_type_2> available at a specific side of an <object>.
--|
--| Parameter Description:
--|   a_<object> is the <object> being acted on.
--|   a_<object>_side is the side queried
--|   <state_type_2> is declared ...
--|*************************************************************

    The_<State_Type_2> : <State_Type_2>;

    begin
        if A_<Object>.<Attribute_2> = Yyyyy then
            The_<State_Type_2> := A_<Object>.Points (
                Opposite_Side (A_<Object>_Side));
        end if;

        RETURN The_<State_Type_2>;

    end Get_<State_Type_2>_From;


pragma Page;
```

```
procedure Give_<Attribute_1>_To (
    A_<Object> : in <Object>;
    <Attribute_1> : in <Object>_<Attribute_1>) is
--|*************************************************************
--| Description:
--|   sets the value of the state <attribute_1> in the record
--|   representing the the <object>
--|
--| Parameter Description:
--|   a_<object> is the <object> whose state is to be updated
--|   <attribute_1> is the new <object> <attribute_1>
--|
--| Notes:
--|*************************************************************

    begin
        A_<Object>.<Attribute_1> := <Attribute_1>;

    end Give_<Attribute_1>_To;


function Get_<Attribute_1>_From (
    A_<Object> : in <Object>
    ) return <Object>_<Attribute_1> is
--|*************************************************************
--| Description:
--|   reads the value of the state <attribute_1> in the record
--|   representing the <object>
--|
--| Parameter Description:
--|   a_<object> is the <object> whose state is read
--|   <attribute_1> is the current <attribute_1>
--|
--| Notes:
--|   none
--|*************************************************************

    begin
        RETURN A_<Object>.<Attribute_1>;

    end Get_<Attribute_1>_From;

pragma Page;
```

```
procedure Give_<Attribute_2>_To (
    A_<Object> : in <Object>;
    <Attribute_2> : in <Object>_<Attribute_2>) is
--|*************************************************************
--| Description:
--|   sets the value of the state <attribute_2> in the record
--|   representing the the <object>
--|
--| Parameter Description:
--|   a_<object> is the <object> whose state is to be updated
--|   <attribute_2> is the new <object> <attribute_2>
--|
--| Notes:
--|*************************************************************

    begin
        A_<Object>.<Attribute_2> := <Attribute_2>;

    end Give_<Attribute_2>_To;


function Get_<Attribute_2>_From (
    A_<Object> : in <Object>
    ) return <Object>_<Attribute_2> is
--|*************************************************************
--| Description:
--|   reads the value of the state <attribute_2> in the record
--|   representing the <object>
--|
--| Parameter Description:
--|   a_<object> is the <object> whose state is read
--|   <attribute_2> is the current <attribute_2>
--|
--| Notes:
--|   none
--|*************************************************************

    begin
        RETURN A_<Object>.<Attribute_2>;

    end Get_<Attribute_2>_From;

end <Object>_Object_Manager;
```

Appendix C: Engine code

The Ada code that follows implements a simulator Engine system. The implementation is complete only through the package specifications. The intent is to demonstrate the software architecture defined by the object paradigm discussed in Chapter 4.

C.1. Package Global_Types

```
-- ****************************************************************
--| Module Name:
--|    Global Types
--|
--| Module Type:
--|    Package Specification
--|
--| Module Purpose:
--|    provide global types for use throughout the simulator code
--|----------------------------------------------------------------
--| Module Description:
--|    This package provides global types for use throughout the simulator
--|        code.  The types include those necessary for compliance with the
--|        Boeing ASVP Ada code.
--|
--|        Type Execution_Sequence defines the frames to be used by the
--|        executives during the cyclic execution of the code.
--|
--| References:
--|    Design Documents:
--|       none
--|
--|    User's Manual:
--|       none
--|
--|    Testing and Validation:
--|       none
--|
--| Notes:
--|    none
--|----------------------------------------------------------------
--| Modification History:
--|       24Apr87   kl  created
--|
--|----------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    TBD
--|
```

```
--| Disclaimer:
--|   "This work was sponsored by the Department of Defense.
--|    The views and conclusions contained in this document are
--|    solely those of the author(s) and should not be interpreted as
--|    representing official policies, either expressed or implied,
--|    of the Software Engineering Institute, Carnegie Mellon University,
--|    the U.S. Air Force, the Department of Defense, or the U.S. Government."
--|*****************************************************************

package Global_Types is

    type Execution_Sequence is (
        Frame_1_Modules_Are_Executed,
        Frame_2_Modules_Are_Executed,
        Frame_3_Modules_Are_Executed,
        Frame_4_Modules_Are_Executed,
        Frame_5_Modules_Are_Executed,
        Frame_6_Modules_Are_Executed,
        Frame_7_Modules_Are_Executed,
        Frame_8_Modules_Are_Executed
        );

end Global_Types;
```

C.2. Package Standard_Engineering_Types

```
--|*****************************************************************
--| Module Name:
--|   Standard_Engineering_Types
--|
--| Module Type:
--|   Package Specification
--|
--| Module Purpose:
--|   This package defines some standard engineering symbols and units
--|  which are used in the Flight_System.
--|-----------------------------------------------------------------
--| Module Description:
--|   The standard engineering symbols, their range and units of measure
--|  are specified in this package.  All objects and types in the
--|  flight_system which are represented in the real world in these units
--|  should be derived from these types.  New derived types can be expressed
--|  as follows:
--|     type My_Blark is new Standard_Engineering_Types.Blark;
--|
--| References:
--|   Design Documents:
--|      none
--|
--|   User's Manual:
--|      none
--|
--|   Testing and Validation:
--|      none
--|
--| Notes:
--|   none
--|-----------------------------------------------------------------
--| Modification History:
--|   25Aug87        cpp    created
--|
--|-----------------------------------------------------------------
--| Distribution and Copyright Notice:
```

```
--|   TBD
--|
--| Disclaimer:
--|   "This work was sponsored by the Department of Defense.
--|   The views and conclusions contained in this document are
--|   solely those of the author(s) and should not be interpreted as
--|   representing official policies, either expressed or implied,
--|   of the Software Engineering Institute, Carnegie Mellon University,
--|   the U.S. Air Force, the Department of Defense, or the U.S. Government."
--|*****************************************************************

package Standard_Engineering_Types is

   type Pressure       is digits 6 range 0.0 .. 10000.0;
                    -- pounds per square inch
   type Temperature    is range 300 .. 3000;
                    -- degrees Rankine
   type Air_Flow       is digits 4 range 0.0  .. 500.0;
                    -- pounds per second
   type Fuel_Flow      is digits 2 range 0.0  .. 5.0;
                    -- pounds per second
   type Thrust         is digits 6 range 0.0 .. 20250.0;
                    -- pounds
   type Rpm            is range 0 .. 20000;
                    -- revolutions per minute
   type Torque         is range 0 .. 10000;
                    -- pound feet

end Standard_Engineering_Types;
```

C.3. Package Bleed_Valve_Object_Manager

```
--|*****************************************************************
--| Module Name:
--|   Bleed_Valve_Object_Manager
--|
--| Module Type:
--|   Package Specification
--|
--| Module Purpose:
--|   This package manages objects which simulate the
--|   Engine Bleed_Valve for the C-141 simulator.
--|   This management entails creation of Engine Bleed_Valve objects,
--|   update and maintenance of its state, and finally state
--|   reporting capabilities.
--|-----------------------------------------------------------------
--| Module Description:
--|   The Engine Bleed_Valve object manager provides a means to create
--|   a Bleed_Valve object via the New_Bleed_Valve entry and returns
--|   an identification for the Bleed_Valve, which is to be used when
--|   updating/accessing the Bleed_Valve objects state as described below.
--|
--|   The Engine Bleed_Valve object manager provides a means to update the
--|   state of the object via the:
--|      1) Give_Inlet_Air_Flow_To
--|      2) Give_Discharge_Pressure_To
--|   entries, requiring the following external state information:
--|      1) Inlet_Air_Flow        pounds per second
--|      2) Discharge_Pressure    pounds per square foot
--|
--|   The Engine Bleed_Valve object manager provides a means of obtaining
--|   state information via the:
```

```
--|     3) Get_Inlet_Pressure_From
--|     4) Get_Discharge_Air_Flow_From
--|  entries, yielding the following internal state information:
--|     3) Inlet_Pressure        pounds per square foot
--|     4) Discharge_Air_Flow    pounds per second
--|
--|  References:
--|   Design Documents:
--|     none
--|
--|   User's Manual:
--|     none
--|
--|   Testing and Validation:
--|     none
--|
--|  Notes:
--|   none
--|
--|------------------------------------------------------------------
--|  Modification History:
--|   27Aug87    cpp    created
--|
--|------------------------------------------------------------------
--|  Distribution and Copyright Notice:
--|   TBD
--|
--|  Disclaimer:
--|   "This work was sponsored by the Department of Defense.
--|   The views and conclusions contained in this document are
--|   solely those of the author(s) and should not be interpreted as
--|   representing official policies, either expressed or implied,
--|   of the Software Engineering Institute, Carnegie Mellon University,
--|   the U.S. Air Force, the Department of Defense, or the U.S. Government."
--|**********************************************************************
--|

with Standard_Engineering_Types;

package Bleed_Valve_Object_Manager is

   type Bleed_Valve is private ;-- a Bleed_Valve is an abstraction of a
                    -- Bleed_Valve within a Engine.

   function New_Bleed_Valve return Bleed_Valve;
   --|*********************************************************************
   --|  Description:
   --|     This function returns a pointer to a new Bleed_Valve object
   --|        representation. This pointer will be used to identify
   --|        the object for state update and state reporting purposes.
   --|
   --|  Parameter Description:
   --|     return Bleed_Valve
   --|           Pointer to a Bleed_Valve object.
   --|*********************************************************************


   procedure Give_Inlet_Air_Flow_To(
     A_Bleed_Valve          : in Bleed_Valve;
     Given_Inlet_Air_Flow   : in Standard_Engineering_Types.Air_Flow
   );
   --|*********************************************************************
   --|     Description:
   --|        Initiates a change in the specified Bleed_Valve object's
   --|        state given the Inlet_Air_Flow.
```

```
--|
--|    Parameter Description:
--|      A_Bleed_Valve
--|        Identifies the Bleed_Valve whose state is to be changed.
--|      Given_Inlet_Air_Flow
--|        Is the Inlet_Air_Flow, in pounds per second,
--|        which is to affect the state of the Bleed_Valve object.
--|*************************************************************************

  procedure Give_Discharge_Pressure_To(
    A_Bleed_Valve           : in Bleed_Valve;
    Given_Discharge_Pressure: in Standard_Engineering_Types.Pressure
  );
--|*************************************************************************
--|    Description:
--|      Initiates a change in the specified Bleed_Valve object's
--|      state given the Discharge_Pressure.
--|
--|    Parameter Description:
--|      A_Bleed_Valve
--|        Identifies the Bleed_Valve whose state is to be changed.
--|      Given_Discharge_Pressure
--|        Is the Discharge_Pressure, in pounds per square foot,
--|        which is to affect the state of the Bleed_Valve object.
--|*************************************************************************


  function Get_Inlet_Pressure_From(
    A_Bleed_Valve           : in Bleed_Valve
    ) return Standard_Engineering_Types.Pressure;
--|*************************************************************************
--| Description:
--|      Initiates a report of the specified Bleed_Valve object's
--|      state returning the Inlet_Pressure.
--|
--|    Parameter Description:
--|      A_Bleed_Valve
--|        Identifies the Bleed_Valve whose state is needed.
--|      return Pressure
--|        Is the Inlet_Pressure portion of Bleed_Valve object's state,
--|        in pounds per square foot, which is to be reported on.
--|*************************************************************************

  function Get_Discharge_Air_Flow_From(
    A_Bleed_Valve           : in Bleed_Valve
    ) return Standard_Engineering_Types.Air_Flow;
--|*************************************************************************
--| Description:
--|      Initiates a report of the specified Bleed_Valve object's
--|      state returning the Discharge_Air_Flow.
--|
--|    Parameter Description:
--|      A_Bleed_Valve
--|        Identifies the Bleed_Valve whose state is needed.
--|      return Air_Flow
--|        Is the Discharge_Air_Flow portion of Bleed_Valve object's state,
--|        in pounds per second, which is to be reported on.
--|*************************************************************************

private
  type Bleed_Valve_Representation; -- incomplete type, defined in
                                   -- package body
  type Bleed_Valve is access Bleed_Valve_Representation;
                                   -- pointer to a Bleed_Valve representation
end Bleed_Valve_Object_Manager;
```

C.4. Package Burner_Object_Manager

```
--|*****************************************************************
--| Module Name:
--|   Burner_Object_Manager
--|
--| Module Type:
--|   Package Specification
--|
--| Module Purpose:
--|   This package manages objects which simulate the
--|   Engine Burner for the C-141 simulator.
--|   This management entails creation of Engine Burner objects,
--|   update and maintenance of its state, and finally state
--|   reporting capabilities.
--|-----------------------------------------------------------------
--| Module Description:
--|   The Engine Burner object manager provides a means to create
--|   a Burner object via the New_Burner entry and returns
--|   an identification for the Burner, which is to be used when
--|   updating/accessing the Burner objects state as described below.
--|
--|   The Engine Burner object manager provides a means to update the
--|   state of the object via the:
--|      1) Give_Inlet_Air_To
--|      2) Give_Fuel_Flow_To
--|      3) Give_Spark_To
--|   entries, requiring the following external state information:
--|      1) Inlet_Pressure     pounds per square inch
--|         Inlet_Temperature  degrees Rankine
--|         Inlet_Air_Flow     pounds per second
--|      2) Fuel_Flow          pounds per second
--|      3) Spark              joules
--|
--|   The Engine Burner object manager provides a means of obtaining
--|   state information via the:
--|      4) Get_Discharge_Air_From
--|   entries, yielding the following internal state information:
--|      4) Discharge_Pressure  pounds per square inch
--|         Discharge_Temperature degrees Rankine
--|         Discharge_Air_Flow  pounds per second
--|
--| References:
--|   Design Documents:
--|      none
--|
--|   User's Manual:
--|      none
--|
--|   Testing and Validation:
--|      none
--|
--| Notes:
--|   none
--|
--|-----------------------------------------------------------------
--| Modification History:
--|   24Aug87    cpp    created
--|
--|-----------------------------------------------------------------
--| Distribution and Copyright Notice:
--|   TBD
--|
--| Disclaimer:
--|   "This work was sponsored by the Department of Defense.
```

```
--    The views and conclusions contained in this document are
--    solely those of the author(s) and should not be interpreted as
--    representing official policies, either expressed or implied,
--    of the Software Engineering Institute, Carnegie Mellon University,
--    the U.S. Air Force, the Department of Defense, or the U.S. Government."
-- ******************************************************************************
--

with Standard_Engineering_Types;

package Burner_Object_Manager is

   type Burner is private ;-- a Burner is an abstraction of a
                           -- Burner within a Engine.

   type Spark                is range (None, Low, High);

   function New_Burner return Burner;
   -- ********************************************************************
   -- | Description:
   -- |    This function returns a pointer to a new Burner object
   -- |        representation. This pointer will be used to identify
   -- |        the object for state update and state reporting purposes.
   -- |
   -- | Parameter Description:
   -- |    return Burner
   -- |           Pointer to a Burner object.
   -- ********************************************************************


   procedure Give_Inlet_Air_To(
     A_Burner           : in Burner;
     Given_Inlet_Pressure   : in Standard_Engineering_Types.Pressure;
     Given_Inlet_Temperature    : in Standard_Engineering_Types.Temperature;
     Given_Inlet_Air_Flow   : in Standard_Engineering_Types.Air_Flow
   );
   -- ********************************************************************
   -- |     Description:
   -- |        Initiates a change in the specified Burner object's
   -- |        state given the Inlet_Pressure, Inlet_Temperature,
   -- |        and the Inlet_Air_Flow.
   -- |
   -- |     Parameter Description:
   -- |        A_Burner
   -- |           Identifies the Burner whose state is to be changed.
   -- |        Given_Inlet_Pressure
   -- |           Is the Inlet_Pressure, in pounds per square inch,
   -- |           which is to affect the state of the Burner object.
   -- |        Given_Inlet_Temperature
   -- |           Is the Inlet_Temperature, in degrees Rankine,
   -- |           which is to affect the state of the Burner object.
   -- |        Given_Inlet_Air_Flow
   -- |           Is the Inlet_Air_Flow, in pounds per second,
   -- |           which is to affect the state of the Burner object.
   -- ********************************************************************

   procedure Get_Discharge_Air_From(
     A_Burner               : in  Burner;
     Returning_Discharge_Pressure   : out Standard_Engineering_Types.Pressure;
     Returning_Discharge_Temperature: out Standard_Engineering_Types.Temperature;
     Returning_Discharge_Air_Flow   : out Standard_Engineering_Types.Air_Flow
   );
   -- ********************************************************************
   -- |     Description:
   -- |        Initiates a report of the specified Burner object's
```

```
--|        state returning the Discharge_Pressure,
--|        Discharge_Temperature, and the Discharge_Air_Flow.
--|
--|     Parameter Description:
--|        A_Burner
--|           Identifies the Burner whose state is needed.
--|        Returning_Discharge_Pressure
--|           Is the Discharge_Pressure portion of Burner object's state,
--|           in pounds per square inch, which is to be reported on.
--|        Returning_Discharge_Temperature
--|           Is the Discharge_Temperature portion of Burner object's state,
--|           in degrees Rankine, which is to be reported on.
--|        Returning_Discharge_Air_Flow
--|           Is the Discharge_Air_Flow portion of Burner object's state,
--|           in pounds per second, which is to be reported on.
--|*********************************************************************


   procedure Give_Fuel_Flow_To(
      A_Burner          : in Burner;
      Given_Fuel_Flow   : in Standard_Engineering_Types.Fuel_Flow
   );
--|*********************************************************************
--|     Description:
--|        Initiates a change in the specified Burner object's
--|        state given the Fuel_Flow.
--|
--|     Parameter Description:
--|        A_Burner
--|           Identifies the Burner whose state is to be changed.
--|        Given_Fuel_Flow
--|           Is the Fuel_Flow, in pounds per second,
--|           which is to affect the state of the Burner object.
--|*********************************************************************

   procedure Give_Spark_To(
      A_Burner          : in Burner;
      Given_Spark       : in Spark
   );
--|*********************************************************************
--|     Description:
--|        Initiates a change in the specified Burner object's
--|        state given the Spark.
--|
--|     Parameter Description:
--|        A_Burner
--|           Identifies the Burner whose state is to be changed.
--|        Given_Spark
--|           Is the Spark, in joules,
--|           which is to affect the state of the Burner object.
--|*********************************************************************

private
   type Burner_Representation;       -- incomplete type, defined in
                                     -- package body
   type Burner is access Burner_Representation;
                                     -- pointer to a Burner representation
end Burner_Object_Manager;
```

C.5. Package Diffuser_Object_Manager

```
--|*****************************************************************
--| Module Name:
--|   Diffuser_Object_Manager
--|
--| Module Type:
--|   Package Specification
--|
--| Module Purpose:
--|   This package manages objects which simulate the
--|   Engine Diffuser for the C-141 simulator.
--|   This management entails creation of Engine Diffuser objects,
--|   update and maintenance of its state, and finally state
--|   reporting capabilities.
--|-----------------------------------------------------------------
--| Module Description:
--|   The Engine Diffuser object manager provides a means to create
--|   a Diffuser object via the New_Diffuser entry and returns
--|   an identification for the Diffuser, which is to be used when
--|   updating/accessing the Diffuser objects state as described below.
--|
--|   The Engine Diffuser object manager provides a means to update the
--|   state of the object via the:
--|      1) Give_Inlet_Air_To
--|      2) Give_Mach_Number_To
--|   entries, requiring the following external state information:
--|      1) Inlet_Pressure        pounds per square foot
--|         Inlet_Temperature     degrees Rankine
--|      2) Mach_Number           <dimensionless>
--|
--|   The Engine Diffuser object manager provides a means of obtaining
--|   state information via the:
--|      3) Get_Discharge_Air_From
--|   entries, yielding the following internal state information:
--|      3) Discharge_Pressure    pounds per square foot
--|         Discharge_Temperature  degrees Rankine
--|         Discharge_Air_Flow    pounds per second
--|
--| References:
--|   Design Documents:
--|      none
--|
--|   User's Manual:
--|      none
--|
--|   Testing and Validation:
--|      none
--|
--| Notes:
--|   none
--|
--|-----------------------------------------------------------------
--| Modification History:
--|   25Aug87    cpp    created
--|
--|-----------------------------------------------------------------
--| Distribution and Copyright Notice:
--|   TBD
--|
--| Disclaimer:
--|   "This work was sponsored by the Department of Defense.
--|    The views and conclusions contained in this document are
--|    solely those of the author(s) and should not be interpreted as
--|    representing official policies, either expressed or implied,
```

```
--  of the Software Engineering Institute, Carnegie Mellon University,
--  the U.S. Air Force, the Department of Defense, or the U.S. Government."
-- ****************************************************************************
--

with Standard_Engineering_Types;

package Diffuser_Object_Manager is

  type Diffuser is private ;     -- a Diffuser is an abstraction of an
                                 -- Diffuser within a Engine.

  type Mach_Number      is digits 3 range 0.00 .. 1.00;
                                 -- <dimensionless>

  function New_Diffuser return Diffuser;
  -- ************************************************************************
  -- | Description:
  -- |   This function returns a pointer to a new Diffuser object
  -- |       representation.  This pointer will be used to identify
  -- |       the object for state update and state reporting purposes.
  -- |
  -- | Parameter Description:
  -- |   return Diffuser
  --           Pointer to a Diffuser object.
  -- ************************************************************************


  procedure Give_Inlet_Pressure_To(
    A_Diffuser          : in Diffuser;
    Given_Inlet_Pressure    : in Standard_Engineering_Types.Pressure;
    Given_Inlet_Temperature    : in Standard_Engineering_Types.Temperature
  );
  -- |**********************************************************************
  -- |   Description:
  -- |     Initiates a change in the specified Diffuser object's
  -- |     state given the Inlet_Pressure, and Inlet_Temperature.
  -- |
  -- |   Parameter Description:
  -- |     A_Diffuser
  -- |       Identifies the Diffuser whose state is to be changed.
  -- |     Given_Inlet_Pressure
  -- |       Is the Inlet_Pressure, in pounds per square foot,
  -- |       which is to affect the state of the Diffuser object.
  -- |     Given_Inlet_Temperature
  -- |       Is the Inlet_Temperature, in degrees Rankine,
  -- |       which is to affect the state of the Diffuser object.
  -- |**********************************************************************

  procedure Give_Mach_Number_To(
    A_Diffuser          : in Diffuser;
    Given_Mach_Number   : in Mach_Number
  );
  -- |**********************************************************************
  -- |   Description:
  -- |     Initiates a change in the specified Diffuser object's
  -- |     state given the Mach_Number.
  -- |
  -- |   Parameter Description:
  -- |     A_Diffuser
  -- |       Identifies the Diffuser whose state is to be changed.
  -- |     Given_Mach_Number
  -- |       Is the Mach_Number, in <dimensionless>,
  -- |       which is to affect the state of the Diffuser object.
  -- |**********************************************************************
```

```
   procedure Get_Discharge_Air_From(
      A_Diffuser                  : in  Diffuser;
      Returning_Discharge_Pressure: out Standard_Engineering_Types.Pressure;
      Returning_Discharge_Temperature: out Standard_Engineering_Types.Temperature;
      Returning_Discharge_Air_Flow   : out Standard_Engineering_Types.Air_Flow
   );
   --|*********************************************************************
   --| Description:
   --|        Initiates a report of the specified Diffuser object's
   --|        state returning the Discharge_Pressure and Discharge_Temperature.
   --|
   --|     Parameter Description:
   --|        A_Diffuser
   --|           Identifies the Diffuser whose state is needed.
   --|        Returning_Discharge_Pressure
   --|           Is the Discharge_Pressure portion of Diffuser object's state,
   --|           in pounds per square foot, which is to be reported on.
   --|        Returning_Discharge_Temperature
   --|           Is the Discharge_Temperature portion of Diffuser object's state,
   --|           in degrees Rankine, which is to be reported on.
   --|        Returning_Discharge_Air_Flow
   --|           Is the Discharge_Air_Flow portion of Diffuser object's state,
   --|           in pounds per second, which is to be reported on.
   --|*********************************************************************

private

   type Diffuser_Representation;      -- incomplete type, defined in
                                      -- package body
   type Diffuser is access Diffuser_Representation;
                                      -- pointer to a Diffuser representation
end Diffuser_Object_Manager;
```

C.6. Package Exhaust_Object_Manager

```
--|*********************************************************************
--| Module Name:
--|   Exhaust_Object_Manager
--|
--| Module Type:
--|   Package Specification
--|
--| Module Purpose:
--|   This package manages objects which simulate the
--|   Engine Exhaust for the C-141 simulator.
--|   This management entails creation of Engine Exhaust objects,
--|   update and maintenance of its state, and finally state
--|   reporting capabilities.
--|---------------------------------------------------------------------
--| Module Description:
--|   The Engine Exhaust object manager provides a means to create
--|   an Exhaust object via the New_Exhaust entry and returns
--|   an identification for the Exhaust, which is to be used when
--|   updating/accessing the Exhaust objects state as described below.
--|
--|   The Engine Exhaust object manager provides a means to update the
--|   state of the object via the:
--|      1) Give_Inlet_Pressure_To
--|   entries, requiring the following external state information:
--|      1) Inlet_Pressure     pounds per square inch
--|
--|   The Engine Exhaust object manager provides a means of obtaining
--|   state information via the:
```

```
--|     2) Get_Discharge_Thrust_From
--|     3) Get_EGT_From
--|     4) Get_EPR_From
--|   entries, yielding the following internal state information:
--|     2) Discharge_Thrust    pounds
--|     3) EGT                 degrees Rankine
--|     4) EPR                 <dimensionless>
--|
--| References:
--|   Design Documents:
--|     none
--|
--|   User's Manual:
--|     none
--|
--|   Testing and Validation:
--|     none
--|
--| Notes:
--|   none
--|
--|------------------------------------------------------------------------
--| Modification History:
--|   25Aug87    cpp    created
--|
--|------------------------------------------------------------------------
--| Distribution and Copyright Notice:
--|   TBD
--|
--| Disclaimer:
--|   "This work was sponsored by the Department of Defense.
--|    The views and conclusions contained in this document are
--|    solely those of the author(s) and should not be interpreted as
--|    representing official policies, either expressed or implied,
--|    of the Software Engineering Institute, Carnegie Mellon University,
--|    the U.S. Air Force, the Department of Defense, or the U.S. Government."
--|********************************************************************************
--|

with Standard_Engineering_Types;

package Exhaust_Object_Manager is

  type Exhaust is private ;     -- an Exhaust is an abstraction of an
                                -- Exhaust within an Engine.

  type Epr              is digits 2 range 1.2 .. 2.3;
                        -- <dimensionless>

  function New_Exhaust return Exhaust;
  --|*********************************************************************
  --| Description:
  --|    This function returns a pointer to a new Exhaust object
  --|        representation.  This pointer will be used to identify
  --|        the object for state update and state reporting purposes.
  --|
  --| Parameter Description:
  --|    return Exhaust
  --|          Pointer to an Exhaust object.
  --|*********************************************************************

  procedure Give_Inlet_Pressure_To(
    A_Exhaust          : in Exhaust;
    Given_Inlet_Pressure   : in Standard_Engineering_Types.Pressure
  );
```

```
--|*********************************************************************
--|    Description:
--|       Initiates a change in the specified Exhaust object's
--|       state given the Inlet_Pressure.
--|
--|    Parameter Description:
--|       A_Exhaust
--|          Identifies the Exhaust whose state is to be changed.
--|       Given_Inlet_Pressure
--|          Is the Inlet_Pressure, in pounds per square inch,
--|          which is to affect the state of the Exhaust object.
--|*********************************************************************


function Get_Discharge_Thrust_From(
   A_Exhaust          : in Exhaust
   ) return Standard_Engineering_Types.Thrust;
--|*********************************************************************
--| Description:
--|       Initiates a report of the specified Exhaust object's
--|       state returning the Discharge_Thrust.
--|
--|    Parameter Description:
--|       A_Exhaust
--|          Identifies the Exhaust whose state is needed.
--|       return Thrust
--|          Is the Discharge_Thrust portion of Exhaust object's state,
--|          in pounds, which is to be reported on.
--|*********************************************************************

function Get_Egt_From(
   A_Exhaust          : in Exhaust
   ) return Standard_Engineering_Types.Temperature;
--|*********************************************************************
--| Description:
--|       Initiates a report of the specified Exhaust object's
--|       state returning the EGT.
--|
--|    Parameter Description:
--|       A_Exhaust
--|          Identifies the Exhaust whose state is needed.
--|       return EGT
--|          Is the EGT portion of Exhaust object's state,
--|          in degrees Rankine, which is to be reported on.
--|*********************************************************************

function Get_Epr_From(
   A_Exhaust          : in Exhaust
   ) return Epr;
--|*********************************************************************
--| Description:
--|       Initiates a report of the specified Exhaust object's
--|       state returning the EPR.
--|
--|    Parameter Description:
--|       A_Exhaust
--|          Identifies the Exhaust whose state is needed.
--|       return EPR
--|          Is the EPR portion of Exhaust object's state,
--|          in <dimensionless>, which is to be reported on.
--|*********************************************************************

private
   type Exhaust_Representation;      -- incomplete type, defined in
                                     -- package body
   type Exhaust is access Exhaust_Representation;
```

```
                                -- pointer to an Exhaust representation
end Exhaust_Object_Manager;
```

C.7. Package Fan_Duct_Object_Manager

```
--|*********************************************************************
--| Module Name:
--|   Fan_Duct_Object_Manager
--|
--| Module Type:
--|   Package Specification
--|
--| Module Purpose:
--|   This package manages objects which simulate the
--|   Engine Fan_Duct for the C-141 simulator.
--|   This management entails creation of Engine Fan_Duct objects,
--|   update and maintenance of its state, and finally state
--|   reporting capabilities.
--|---------------------------------------------------------------
--| Module Description:
--|   The Engine Fan_Duct object manager provides a means to create
--|   a Fan_Duct object via the New_Fan_Duct entry and returns
--|   an identification for the Fan_Duct, which is to be used when
--|   updating/accessing the Fan_Duct objects state as described below.
--|
--|   The Engine Fan_Duct object manager provides a means to update the
--|   state of the object via the:
--|      1) Give_Inlet_Pressure_To
--|   entries, requiring the following external state information:
--|      1) Inlet_Pressure     pounds per square inch
--|
--|   The Engine Fan_Duct object manager provides a means of obtaining
--|   state information via the.
--|      2) Get_Discharge_Thrust_From
--|   entries, yielding the following internal state information:
--|      2) Discharge_Thrust    pounds
--|
--| References:
--|   Design Documents:
--|      none
--|
--|   User's Manual:
--|      none
--|
--|   Testing and Validation:
--|      none
--|
--| Notes:
--|   none
--|
--|---------------------------------------------------------------
--| Modification History:
--|   25Aug87    cpp    created
--|
--|---------------------------------------------------------------
--| Distribution and Copyright Notice:
--|   TBD
--|
--| Disclaimer:
--|   "This work was sponsored by the Department of Defense.
--|    The views and conclusions contained in this document are
--|    solely those of the author(s) and should not be interpreted as
--|    representing official policies, either expressed or implied,
```

```
--   of the Software Engineering Institute, Carnegie Mellon University,
--   the U.S. Air Force, the Department of Defense, or the U.S. Government."
-- ********************************************************************************
--

with Standard_Engineering_Types;

package Fan_Duct_Object_Manager is

   type Fan_Duct is private ;   -- a Fan_Duct is an abstraction of a
                                -- Fan_Duct within a Engine.


   function New_Fan_Duct return Fan_Duct;
   -- |*********************************************************************
   -- | Description:
   -- |    This function returns a pointer to a new Fan_Duct object
   -- |       representation. This pointer will be used to identify
   -- |       the object for state update and state reporting purposes.
   -- |
   -- | Parameter Description:
   -- |    return Fan_Duct
   -- |          Pointer to a Fan_Duct object.
   -- |*********************************************************************


   procedure Give_Inlet_Pressure_To(
     A_Fan_Duct         : in Fan_Duct;
     Given_Inlet_Pressure   : in Standard_Engineering_Types.Pressure
   );
   -- |*********************************************************************
   -- |    Description:
   -- |       Initiates a change in the specified Fan_Duct object's
   -- |       state given the Inlet_Pressure.
   -- |
   -- |    Parameter Description:
   -- |       A_Fan_Duct
   -- |          Identifies the Fan_Duct whose state is to be changed.
   -- |       Given_Inlet_Pressure
   -- |          Is the Inlet_Pressure, in pounds per square inch,
   -- |          which is to affect the state of the Fan_Duct object.
   -- |*********************************************************************

   function Get_Discharge_Thrust_From(
     A_Fan_Duct         : in Fan_Duct
     ) return Standard_Engineering_Types.Thrust;
   -- |*********************************************************************
   -- | Description:
   -- |       Initiates a report of the specified Fan_Duct object's
   -- |       state returning the Discharge_Thrust.
   -- |
   -- |    Parameter Description:
   -- |       A_Fan_Duct
   -- |          Identifies the Fan_Duct whose state is needed.
   -- |       return Thrust
   -- |          Is the Discharge_Thrust portion of Fan_Duct object's state,
   -- |          in pounds, which is to be reported on.
   -- |*********************************************************************

private
   type Fan_Duct_Representation;    -- incomplete type, defined in
                                    -- package body
   type Fan_Duct is access Fan_Duct_Representation;
                                    -- pointer to a Fan_Duct representation
end Fan_Duct_Object_Manager;
```

C.8. Package Rotor1_Object_Manager

```
--|*****************************************************************
--| Module Name:
--|   Rotor1_Object_Manager
--|
--| Module Type:
--|   Package Specification
--|
--| Module Purpose:
--|   This package manages objects which simulate the
--|   Engine Rotor1 for the C-141 simulator.
--|   This management entails creation of Engine Rotor1 objects,
--|   update and maintenance of its state, and finally state
--|   reporting capabilities.
--|-----------------------------------------------------------------
--| Module Description:
--|   The Engine Rotor1 object manager provides a means to create
--|   a Rotor1 object via the New_Rotor1 entry and returns
--|   an identification for the Rotor1, which is to be used when
--|   updating/accessing the Rotor1 objects state as described below.
--|
--|   The Engine Rotor1 object manager provides a means to update the
--|   state of the object via the:
--|      1) Give_Fan1_Inlet_Air_To
--|      2) Give_Turbine1_Inlet_Air_To
--|   entries, requiring the following external state information:
--|      1) Fan1_Inlet_Pressure       pounds per square inch
--|         Fan1_Inlet_Temperature    degrees Rankine
--|         Fan1_Inlet_Air_Flow       pounds per second
--|      2) Turbine1_Inlet_Pressure   pounds per square foot
--|         Turbine1_Inlet_Temperature degrees Rankine
--|         Turbine1_Inlet_Air_Flow   pounds per second
--|
--|   The Engine Rotor1 object manager provides a means of obtaining
--|   state information via the:
--|      3) Get_Fan1_Discharge_Air_From
--|      4) Get_Turbine1_Discharge_Air_From
--|      5) Get_RPM_From
--|      6) Get_Vibration_From
--|   entries, yielding the following internal state information:
--|      3) Fan1_Discharge_Pressure       pounds per square inch
--|         Fan1_Discharge_Temperature    degrees Rankine
--|         Fan1_Discharge_Air_Flow       pounds per second
--|      4) Turbine1_Discharge_Pressure   pounds per square foot
--|         Turbine1_Discharge_Temperature degrees Rankine
--|         Turbine1_Discharge_Air_Flow   pounds per second
--|      5) RPM                           rpm
--|      6) Vibration                     mils
--|
--| References:
--|   Design Documents:
--|      none
--|
--|   User's Manual:
--|      none
--|
--|   Testing and Validation:
--|      none
--|
--| Notes:
--|   none
--|
--|-----------------------------------------------------------------
--| Modification History:
```

```
--| 25Aug87   cpp   created
--|
--|-----------------------------------------------------------------
--| Distribution and Copyright Notice:
--|  TBD
--|
--| Disclaimer:
--|  "This work was sponsored by the Department of Defense.
--|  The views and conclusions contained in this document are
--|  solely those of the author(s) and should not be interpreted as
--|  representing official policies, either expressed or implied,
--|  of the Software Engineering Institute, Carnegie Mellon University,
--|  the U.S. Air Force, the Department of Defense, or the U.S. Government."
--|*********************************************************************************
--|


with Standard_Engineering_Types;

package Rotor1_Object_Manager is

  type Rotor1 is private ; -- a Rotor1 is an abstraction of a
                           -- Rotor1 within a Engine.

  type Vibration      is range 0..5;
                      -- mils

  function New_Rotor1 return Rotor1;
  --|*********************************************************************
  --| Description:
  --|   This function returns a pointer to a new Rotor1 object
  --|       representation.  This pointer will be used to identify
  --|       the object for state update and state reporting purposes.
  --|
  --| Parameter Description:
  --|   return Rotor1
  --|         Pointer to a Rotor1 object.
  --|*********************************************************************


  procedure Give_Fan1_Inlet_Air_To(
    A_Rotor1              : in Rotor1;
    Given_Fan1_Inlet_Pressure : in Standard_Engineering_Types.Pressure;
    Given_Fan1_Inlet_Temperature  : in Standard_Engineering_Types.Temperature;
    Given_Fan1_Inlet_Air_Flow : in Standard_Engineering_Types.Air_Flow
  );
  --|*********************************************************************
  --|    Description:
  --|       Initiates a change in the specified Rotor1 object's
  --|       state given the Fan1_Inlet_Pressure, Fan1_Inlet_Temperature,
  --|       and the Fan1_Inlet_Air_Flow.
  --|
  --|    Parameter Description:
  --|       A_Rotor1
  --|          Identifies the Rotor1 whose state is to be changed.
  --|       Given_Fan1_Inlet_Pressure
  --|          Is the Fan1_Inlet_Pressure, in pounds per square inch,
  --|          which is to affect the state of the Rotor1 object.
  --|       Given_Fan1_Inlet_Temperature
  --|          Is the Fan1_Inlet_Temperature, in degrees Rankine,
  --|          which is to affect the state of the Rotor1 object.
  --|       Given_Fan1_Inlet_Air_Flow
  --|          Is the Fan1_Inlet_Air_Flow, in pounds per second,
  --|          which is to affect the state of the Rotor1 object.
  --|*********************************************************************
```

```
procedure Get_Fan1_Discharge_Air_From(
  A_Rotor1            : in  Rotor1;
  Returning_Fan1_Discharge_Pressure  : out Standard_Engineering_Types.Pressure;
  Returning_Fan1_Discharge_Temperature: out Standard_Engineering_Types.Temperature;
  Returning_Fan1_Discharge_Air_Flow  : out Standard_Engineering_Types.Air_Flow
);
--|*****************************************************************
--|    Description:
--|      Initiates a report of the specified Rotor1 object's
--|      state returning the Fan1_Discharge_Pressure,
--|      Fan1_Discharge_Temperature, and the Fan1_Discharge_Air_Flow.
--|
--|    Parameter Description:
--|      A_Rotor1
--|        Identifies the Rotor1 whose state is needed.
--|      Returning_Fan1_Discharge_Pressure
--|        Is the Fan1_Discharge_Pressure portion of Rotor1 object's state,
--|        in pounds per square inch, which is to be reported on.
--|      Returning_Fan1_Discharge_Temperature
--|        Is the Fan1_Discharge_Temperature portion of Rotor1 object's state,
--|        in degrees Rankine, which is to be reported on.
--|      Returning_Fan1_Discharge_Air_Flow
--|        Is the Fan1_Discharge_Air_Flow portion of Rotor1 object's state,
--|        in pounds per second, which is to be reported on.
--|*****************************************************************


procedure Give_Turbine1_Inlet_Air_To(
  A_Rotor1            : in Rotor1;
  Given_Turbine1_Inlet_Pressure  : in Standard_Engineering_Types.Pressure;
  Given_Turbine1_Inlet_Temperature: in Standard_Engineering_Types.Temperature;
  Given_Turbine1_Inlet_Air_Flow  : in Standard_Engineering_Types.Air_Flow
);
--|*****************************************************************
--|    Description:
--|      Initiates a change in the specified Rotor1 object's
--|      state given the Turbine1_Inlet_Pressure, Turbine1_Inlet_Temperature,
--|      and the Turbine1_Inlet_Air_Flow.
--|
--|    Parameter Description:
--|      A_Rotor1
--|        Identifies the Rotor1 whose state is to be changed.
--|      Given_Turbine1_Inlet_Pressure
--|        Is the Turbine1_Inlet_Pressure, in pounds per square inch,
--|        which is to affect the state of the Rotor1 object.
--|      Given_Turbine1_Inlet_Temperature
--|        Is the Turbine1_Inlet_Temperature, in degrees Rankine,
--|        which is to affect the state of the Rotor1 object.
--|      Given_Turbine1_Inlet_Air_Flow
--|        Is the Turbine1_Inlet_Air_Flow, in pounds per second,
--|        which is to affect the state of the Rotor1 object.
--|*****************************************************************

procedure Get_Turbine1_Discharge_Air_From(
  A_Rotor1            : in  Rotor1;
  Returning_Turbine1_Discharge_Pressure  : out Standard_Engineering_Types.Pressure;
  Returning_Turbine1_Discharge_Temperature: out Standard_Engineering_Types.Temperature;
  Returning_Turbine1_Discharge_Air_Flow  : out Standard_Engineering_Types.Air_Flow
);
--|*****************************************************************
--|    Description:
--|      Initiates a report of the specified Rotor1 object's
--|      state returning the Turbine1_Discharge_Pressure,
--|      Turbine1_Discharge_Temperature, and the Turbine1_Discharge_Air_Flow.
--|
--|    Parameter Description:
```

```
--|        A_Rotor1
--|          Identifies the Rotor1 whose state is needed.
--|        Returning_Turbine1_Discharge_Pressure
--|          Is the Turbine1_Discharge_Pressure portion of Rotor1 object's state,
--|          in pounds per square inch, which is to be reported on.
--|        Returning_Turbine1_Discharge_Temperature
--|          Is the Turbine1_Discharge_Temperature portion of Rotor1 object's state,
--|          in degrees Rankine, which is to be reported on.
--|        Returning_Turbine1_Discharge_Air_Flow
--|          Is the Turbine1_Discharge_Air_Flow portion of Rotor1 object's state,
--|          in pounds per second, which is to be reported on.
--|*************************************************************************

  function Get_Rpm_From(
    A_Rotor1      : in  Rotor1
    ) return  Standard_Engineering_Types.Rpm;
--|*************************************************************************
--| Description:
--|        Initiates a report of the specified Rotor1 object's
--|        state returning the RPM.
--|
--|      Parameter Description:
--|        A_Rotor1
--|          Identifies the Rotor1 whose state is needed.
--|        return RPM
--|          Is the RPM portion of Rotor1 object's state,
--|          in rpm, which is to be reported on.
--|*************************************************************************

  function Get_Vibration_From(
    A_Rotor1      : in  Rotor1
    ) return  Vibration;
--|*************************************************************************
--| Description:
--|        Initiates a report of the specified Rotor1 object's
--|        state returning the Vibration.
--|
--|      Parameter Description:
--|        A_Rotor1
--|          Identifies the Rotor1 whose state is needed.
--|        return Vibration
--|          Is the Vibration portion of Rotor1 object's state,
--|          in mils, which is to be reported on.
--|*************************************************************************

private
  type Rotor1_Representation;        -- incomplete type, defined in
                                     -- package body
  type Rotor1 is access Rotor1_Representation;
                                     -- pointer to a Rotor1 representation
end Rotor1_Object_Manager;
```

C.9. Package Rotor2_Object_Manager

```
--|*************************************************************************
--| Module Name:
--|   Rotor2_Object_Manager
--|
--| Module Type:
--|   Package Specification
--|
--| Module Purpose:
```

```
--|  This package manages objects which simulate the
--|  Engine Rotor2 for the C-141 simulator.
--|  This management entails creation of Engine Rotor2 objects,
--|  update and maintenance of its state, and finally state
--|  reporting capabilities.
--|-------------------------------------------------------------
--| Module Description:
--|  The Engine Rotor2 object manager provides a means to create
--|  a Rotor2 object via the New_Rotor2 entry and returns
--|  an identification for the Rotor2, which is to be used when
--|  updating/accessing the Rotor2 objects state as described below.
--|
--|  The Engine Rotor2 object manager provides a means to update the
--|  state of the object via the:
--|     1) Give_Fan2_Inlet_Air_To
--|     2) Give_Turbine2_Inlet_Air_To
--|     3) Give_Torque_To
--|  entries, requiring the following external state information:
--|     1) Fan2_Inlet_Pressure       pounds per square inch
--|        Fan2_Inlet_Temperature    degrees Rankine
--|        Fan2_Inlet_Air_Flow       pounds per second
--|     2) Turbine2_Inlet_Pressure   pounds per square foot
--|        Turbine2_Inlet_Temperature degrees Rankine
--|        Turbine2_Inlet_Air_Flow   pounds per second
--|     3) Torque                    pound feet
--|
--|  The Engine Rotor2 object manager provides a means of obtaining
--|  state information via the:
--|     4) Get_Fan2_Discharge_Air_From
--|     5) Get_Turbine2_Discharge_Air_From
--|     6) Get_RPM_From
--|     7) Get_Vibration_From
--|  entries, yielding the following internal state information:
--|     3) Fan2_Discharge_Pressure       pounds per square inch
--|        Fan2_Discharge_Temperature    degrees Rankine
--|        Fan2_Discharge_Air_Flow       pounds per second
--|     4) Turbine2_Discharge_Pressure   pounds per square foot
--|        Turbine2_Discharge_Temperature degrees Rankine
--|        Turbine2_Discharge_Air_Flow   pounds per second
--|     5) RPM                           rpm
--|     6) Vibration                     mils
--|
--| References:
--|  Design Documents:
--|     none
--|
--|  User's Manual:
--|     none
--|
--|  Testing and Validation:
--|     none
--|
--| Notes:
--|  none
--|
--|-------------------------------------------------------------
--| Modification History:
--|  25Aug87   cpp   created
--|
--|-------------------------------------------------------------
--| Distribution and Copyright Notice:
--|  TBD
--|
--| Disclaimer:
--|  "This work was sponsored by the Department of Defense.
--|   The views and conclusions contained in this document are
```

```
--|  solely those of the author(s) and should not be interpreted as
--|  representing official policies, either expressed or implied,
--|  of the Software Engineering Institute, Carnegie Mellon University,
--|  the U.S. Air Force, the Department of Defense, or the U.S. Government."
--|*****************************************************************************
--|

with Standard_Engineering_Types;

package Rotor2_Object_Manager is

   type Rotor2 is private ; -- a Rotor2 is an abstraction of an
                            -- Rotor2 within a Engine.

   type Vibration       is range 0..5;
                            -- mils

   function New_Rotor2 return Rotor2;
   --|*********************************************************************
   --| Description:
   --|    This function returns a pointer to a new Rotor2 object
   --|       representation. This pointer will be used to identify
   --|       the object for state update and state reporting purposes.
   --|
   --| Parameter Description:
   --|    return Rotor2
   --|          Pointer to a Rotor2 object.
   --|*********************************************************************


   procedure Give_Fan2_Inlet_Air_To(
     A_Rotor2              : in Rotor2;
     Given_Fan2_Inlet_Pressure : in Standard_Engineering_Types.Pressure;
     Given_Fan2_Inlet_Temperature  : in Standard_Engineering_Types.Temperature;
     Given_Fan2_Inlet_Air_Flow : in Standard_Engineering_Types.Air_Flow
   );
   --|*********************************************************************
   --|    Description:
   --|       Initiates a change in the specified Rotor2 object's
   --|       state given the Fan2_Inlet_Pressure, Fan2_Inlet_Temperature,
   --|       and the Fan2_Inlet_Air_Flow.
   --|
   --|    Parameter Description:
   --|       A_Rotor2
   --|          Identifies the Rotor2 whose state is to be changed.
   --|       Given_Fan2_Inlet_Pressure
   --|          Is the Fan2_Inlet_Pressure, in pounds per square inch,
   --|          which is to affect the state of the Rotor2 object.
   --|       Given_Fan2_Inlet_Temperature
   --|          Is the Fan2_Inlet_Temperature, in degrees Rankine,
   --|          which is to affect the state of the Rotor2 object.
   --|       Given_Fan2_Inlet_Air_Flow
   --|          Is the Fan2_Inlet_Air_Flow, in pounds per second,
   --|          which is to affect the state of the Rotor2 object.
   --|*********************************************************************

   procedure Get_Fan2_Discharge_Air_From(
     A_Rotor2              : in  Rotor2;
     Returning_Fan2_Discharge_Pressure  : out Standard_Engineering_Types.Pressure;
     Returning_Fan2_Discharge_Temperature: out Standard_Engineering_Types.Temperature;
     Returning_Fan2_Discharge_Air_Flow  : out Standard_Engineering_Types.Air_Flow
   );
   --|*********************************************************************
   --|    Description:
   --|       Initiates a report of the specified Rotor2 object's
```

```
--|        state returning the Fan2_Discharge_Pressure,
--|        Fan2_Discharge_Temperature, and the Fan2_Discharge_Air_Flow.
--|
--|     Parameter Description:
--|        A_Rotor2
--|           Identifies the Rotor2 whose state is needed.
--|        Returning_Fan2_Discharge_Pressure
--|           Is the Fan2_Discharge_Pressure portion of Rotor2 object's state,
--|           in pounds per square inch, which is to be reported on.
--|        Returning_Fan2_Discharge_Temperature
--|           Is the Fan2_Discharge_Temperature portion of Rotor2 object's state,
--|           in degrees Rankine, which is to be reported on.
--|        Returning_Fan2_Discharge_Air_Flow
--|           Is the Fan2_Discharge_Air_Flow portion of Rotor2 object's state,
--|           in pounds per second, which is to be reported on.
--|*****************************************************************

procedure Give_Turbine2_Inlet_Air_To(
   A_Rotor2                  : in Rotor2;
   Given_Turbine2_Inlet_Pressure  : in Standard_Engineering_Types.Pressure;
   Given_Turbine2_Inlet_Temperature: in Standard_Engineering_Types.Temperature;
   Given_Turbine2_Inlet_Air_Flow  : in Standard_Engineering_Types.Air_Flow
);
--|*****************************************************************
--|     Description:
--|        Initiates a change in the specified Rotor2 object's
--|        state given the Turbine2_Inlet_Pressure, Turbine2_Inlet_Temperature,
--|        and the Turbine2_Inlet_Air_Flow.
--|
--|     Parameter Description:
--|        A_Rotor2
--|           Identifies the Rotor2 whose state is to be changed.
--|        Given_Turbine2_Inlet_Pressure
--|           Is the Turbine2_Inlet_Pressure, in pounds per square inch,
--|           which is to affect the state of the Rotor2 object.
--|        Given_Turbine2_Inlet_Temperature
--|           Is the Turbine2_Inlet_Temperature, in degrees Rankine,
--|           which is to affect the state of the Rotor2 object.
--|        Given_Turbine2_Inlet_Air_Flow
--|           Is the Turbine2_Inlet_Air_Flow, in pounds per second,
--|           which is to affect the state of the Rotor2 object.
--|*****************************************************************

procedure Get_Turbine2_Discharge_Air_From(
   A_Rotor2                  : in  Rotor2;
   Returning_Turbine2_Discharge_Pressure  : out Standard_Engineering_Types.Pressure;
   Returning_Turbine2_Discharge_Temperature: out Standard_Engineering_Types.Temperature;
   Returning_Turbine2_Discharge_Air_Flow  : out Standard_Engineering_Types.Air_Flow
);
--|*****************************************************************
--|     Description:
--|        Initiates a report of the specified Rotor2 object's
--|        state returning the Turbine2_Discharge_Pressure,
--|        Turbine2_Discharge_Temperature, and the Turbine2_Discharge_Air_Flow.
--|
--|     Parameter Description:
--|        A_Rotor2
--|           Identifies the Rotor2 whose state is needed.
--|        Returning_Turbine2_Discharge_Pressure
--|           Is the Turbine2_Discharge_Pressure portion of Rotor2 object's state,
--|           in pounds per square inch, which is to be reported on.
--|        Returning_Turbine2_Discharge_Temperature
--|           Is the Turbine2_Discharge_Temperature portion of Rotor2 object's state,
--|           in degrees Rankine, which is to be reported on.
--|        Returning_Turbine2_Discharge_Air_Flow
```

```
--|          Is the Turbine2_Discharge_Air_Flow portion of Rotor2 object's state,
--|          in pounds per second, which is to be reported on.
--|*****************************************************************

  function Get_Rpm_From(
    A_Rotor2       : in  Rotor2
    ) return  Standard_Engineering_Types.Rpm;
--|*****************************************************************
--| Description:
--|       Initiates a report of the specified Rotor2 object's
--|       state returning the RPM.
--|
--|    Parameter Description:
--|       A_Rotor2
--|          Identifies the Rotor2 whose state is needed.
--|       return RPM
--|          Is the RPM portion of Rotor2 object's state,
--|          in rpm, which is to be reported on.
--|*****************************************************************

  function Get_Vibration_From(
    A_Rotor2       : in  Rotor2
    ) return  Vibration;
--|*****************************************************************
--| Description:
--|       Initiates a report of the specified Rotor2 object's
--|       state returning the Vibration.
--|
--|    Parameter Description:
--|       A_Rotor2
--|          Identifies the Rotor2 whose state is needed.
--|       return Vibration
--|          Is the Vibration portion of Rotor2 object's state,
--|          in mils, which is to be reported on.
--|*****************************************************************

  procedure Give_Torque_To(
    A_Rotor2       : in  Rotor2;
    Given_Torque       : in  Standard_Engineering_Types.Torque
  );
--|*****************************************************************
--|    Description:
--|       Initiates a change in the specified Rotor2 object's
--|       state given the Torque.
--|
--|    Parameter Description:
--|       A_Rotor2
--|          Identifies the Rotor2 whose state is to be changed.
--|       Given_Torque
--|          Is the Torque, in pound feet,
--|          which is to affect the state of the Rotor2 object.
--|*****************************************************************

private
  type  Rotor2_Representation;          -- incomplete type, defined in
                                        -- package body
  type  Rotor2 is access  Rotor2_Representation;
                                        -- pointer to a Rotor2 representation
end  Rotor2_Object_Manager;
```

C.10. Package Flight_Systems

```
--|*****************************************************************
--| Module Name:
--|    Flight Systems
--|
--| Module Type:
--|    Package Specification
--|
--| Module Purpose:
--|    Executive for flight systems
--|-----------------------------------------------------------------
--| Module Description:
--|    This executive is responsible for processing all flight systems.
--|    Processing involves handling all connections between the flight
--|    systems and processing each system.
--|
--| References:
--|    Design Documents:
--|       none
--|
--|    User's Manual:
--|       none
--|
--|    Testing and Validation:
--|       none
--|
--| Notes:
--|    none
--|-----------------------------------------------------------------
--| Modification History:
--|    21Aug87   kl   created
--|
--|-----------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    TBD
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense.
--|    The views and conclusions contained in this document are
--|    solely those of the author(s) and should not be interpreted as
--|    representing official policies, either expressed or implied,
--|    of the Software Engineering Institute, Carnegie Mellon University,
--|    the U.S. Air Force, the Department of Defense, or the U.S. Government."
--|*****************************************************************

with Global_Types;
use Global_Types;

package Flight_Systems is

procedure Update_Flight_Systems (Frame: in Global_Types.Execution_Sequence);
--|*****************************************************************
--| Description:
--|    executive which updates all flight systems
--|
--| Parameter Description:
--|    frame is the current executing frame
--|*****************************************************************

end Flight_Systems;
```

C.11. Package body Flight_Systems

```
--|*********************************************************************
--| Module Name:
--|    Flight Systems
--|
--| Module Type:
--|    Package Body
--|
--|---------------------------------------------------------------------
--| Module Description:
--|    This executive is responsible for processing all flight systems.
--|    Processing involves handling all connections between the flight
--|    systems and processing each system.
--|
--|
--| References:
--|    Design Documents:
--|       none
--|
--|    Testing and Validation:
--|       none
--|
--| Notes:
--|    none
--|---------------------------------------------------------------------
--| Modification History:
--|    21Aug87   kl   created
--|
--|---------------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    TBD
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense.
--|    The views and conclusions contained in this document are
--|    solely those of the author(s) and should not be interpreted as
--|    representing official policies, either expressed or implied,
--|    of the Software Engineering Institute, Carnegie Mellon University,
--|    the U.S. Air Force, the Department of Defense, or the U.S. Government."
--|*********************************************************************


with Flight_Systems_Connection_Manager;
with Flight_Subsystem_Names; use Flight_Subsystem_Names;

with Engine_Updater;
with System_Power_Updater;

package body Flight_Systems is

    type Active_In_Frame is array (Name_Of_A_Flight_Subsystem)
        of Boolean;

    Its_Time_To_Do : constant array (Global_Types.Execution_Sequence) of
                Active_In_Frame:=
      (Frame_1_Modules_Are_Executed => (Engine_1 => (True),
                          others => (False)),
       Frame_2_Modules_Are_Executed => (Ac_Power => (True),
                          others => (False)),
       Frame_3_Modules_Are_Executed => (Engine_2 => (True),
                          others => (False)),
       Frame_4_Modules_Are_Executed => (Dc_Power => (True),
                          others => (False)),
       Frame_5_Modules_Are_Executed => (Engine_3 => (True),
```

```
                        others => (False)),
        Frame_6_Modules_Are_Executed => (others => (False)),
        Frame_7_Modules_Are_Executed => (Engine_4 => (True),
                        others => (False)),
        Frame_8_Modules_Are_Executed => (others => (False))
        );


procedure Update_Flight_Systems (Frame: in Global_Types.Execution_Sequence) is
--| ****************************************************************
--| Description:
--|    flight systems executive.  Performs process connections and update
--|    as an atomic action for each subsystem.
--|
--| Parameter Description:
--|    frame is the current executing frame
--|
--| Notes:
--|    none
--| ****************************************************************

    begin
        for A_Subsystem in Name_Of_A_Flight_Subsystem loop
            if Its_Time_To_Do (Frame)(A_Subsystem) then
                case A_Subsystem is

                    when Dc_Power..Ac_Power =>
                        Flight_Systems_Connection_Manager.
                            Process_Power_Connections_To (A_Subsystem);
                        System_Power_Updater.
                            Update_System_Power(A_Subsystem);

                    when Engine_1..Engine_4 =>
                        Flight_Systems_Connection_Manager.
                            Process_Engine_Connections_To (A_Subsystem);
                        Engine_Updater.
                            Update_Engine (A_Subsystem);
                end case;
            end if;
        end loop;

end Update_Flight_Systems;


begin       -- flight_systems
  null;

end Flight_Systems;
```

C.12. Package Flight_Subsystem_Names

```
--| ****************************************************************
--| Module Name:
--|    Flight Subsystem Names
--|
--| Module Type:
--|    Package Specification
--|
--| Module Purpose:
--|    Names all subsystems under flight systems
--| ----------------------------------------------------------------
```

```
--| Module Description:
--|    Provides the names of all subsystems under flight systems.  The
--|    subsystems are contained in systems, e.g., system power and engines,
--|    under the scope of flight systems.
--|
--| References:
--|    Design Documents:
--|       none
--|
--|    User's Manual:
--|       none
--|
--|    Testing and Validation:
--|       none
--|
--| Notes:
--|    none
--|-----------------------------------------------------------------
--| Modification History:
--|    21Aug87   kl   created
--|
--|-----------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    TBD
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense.
--|    The views and conclusions contained in this document are
--|    solely those of the author(s) and should not be interpreted as
--|    representing official policies, either expressed or implied,
--|    of the Software Engineering Institute, Carnegie Mellon University,
--|    the U.S. Air Force, the Department of Defense, or the U.S. Government."
--|*****************************************************************

package Flight_Subsystem_Names is

   type Name_Of_A_Flight_Subsystem is (Dc_Power, Ac_Power,
         Engine_1, Engine_2, Engine_3, Engine_4);

end Flight_Subsystem_Names;
```

C.13. Package Flight_Systems_Connection_Manager

```
--|*****************************************************************
--| Module Name:
--|    Flight Systems Connection Manager
--|
--| Module Type:
--|    Package Specification
--|
--| Module Purpose:
--|    Describes and processes all connections between flight systems
--|-----------------------------------------------------------------
--| Module Description:
--|    This package is responsible for proccessing all connections between
--|    systems at all levels lower than Flight Systems.
--|
--| References:
--|    Design Documents:
--|       none
--|
--|    User's Manual:
```

```
--|       none
--|
--|    Testing and Validation:
--|       none
--|
--| Notes:
--|    none
--|-----------------------------------------------------------------
--| Modification History:
--|    21Aug87   kl   created
--|
--|-----------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    TBD
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense.
--|    The views and conclusions contained in this document are
--|    solely those of the author(s) and should not be interpreted as
--|    representing official policies, either expressed or implied,
--|    of the Software Engineering Institute, Carnegie Mellon University,
--|    the U.S. Air Force, the Department of Defense, or the U.S. Government."
--| ****************************************************************

with Flight_Subsystem_Names; use Flight_Subsystem_Names;

package Flight_Systems_Connection_Manager is

procedure Process_Power_Connections_To (
     A_Subsystem: in Name_Of_A_Flight_Subsystem);
--| ****************************************************************
--| Description:
--|    This procedure processes all connections between the system power
--|    subsystem and the other systems at the flight executive level.
--|    Processing of connections means to make the subsystem consistent with
--|    its environment.
--|
--| Parameter Description:
--|    a_subsystem is the subsystem to update
--| ****************************************************************

procedure Process_Engine_Connections_To (
     A_Subsystem: in Name_Of_A_Flight_Subsystem);
--| ****************************************************************
--| Description:
--|    This procedure processes all connections between the engine
--|    subsystem and the other systems at the flight executive level.  Processing
--|    of connections means to make the subsystem consistent with its
--|    environment.
--
--| Parameter Description:
--    a_subsystem is the subsystem to update
-- ****************************************************************


end Flight_Systems_Connection_Manager;
```

C.14. Package body Flight_Systems_Connection_Manager

```
--|*****************************************************************
--| Module Name:
--|    Flight Systems Connection Manager
--|
--| Module Type:
--|    Package Body
--|
--|-----------------------------------------------------------------
--| Module Description:
--|    The procedure below defines all connections for passing data
--|    between flight systems.  Each connection is handled by a procedure
--|    call.
--|
--|
--| References:
--|    Design Documents:
--|       none
--|
--|    Testing and Validation:
--|       none
--|
--| Notes:
--|    none
--|-----------------------------------------------------------------
--| Modification History:
--|    21Aug87   kl   created
--|
--|-----------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    TBD
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense.
--|    The views and conclusions contained in this document are
--|    solely those of the author(s) and should not be interpreted as
--|    representing official policies, either expressed or implied,
--|    of the Software Engineering Institute, Carnegie Mellon University,
--|    the U.S. Air Force, the Department of Defense, or the U.S. Government."
--|*****************************************************************


with Engine_Aggregate;
with Rotor2_Object_Manager;
with Ac_Power_Aggregate; use Ac_Power_Aggregate;

package body Flight_Systems_Connection_Manager is

    Engines_To_Idg_Map : array (Engine_1..Engine_4) of
        Ac_Power_Aggregate.Integrated_Drive_Names := (
            Engine_1 => Idg_1,
            Engine_2 => Idg_2,
            Engine_3 => Idg_3,
            Engine_4 => Idg_4);


procedure Process_Power_Connections_To (
        A_Subsystem: in Name_Of_A_Flight_Subsystem) is separate ;
--|*****************************************************************
--| Description:
--|    This procedure processes all connections between the system power
--|    subsystem and the other systems at the flight executive level.
--|    Processing of connections means to make the subsystem consistent with
--|    its environment.
```

```
--|
--| Parameter Description:
--|    a_subsystem is the subsystem to update
--|
--| Notes:
--|    none
--|*********************************************************************

procedure Process_Engine_Connections_To (
          A_Subsystem: in Flight_Subsystem_Names.Name_Of_A_Flight_Subsystem)
              is separate ;
--|*********************************************************************
--| Description:
--|    This procedure processes all connections between the engine
--|    subsystem and the other systems at the flight executive level.  Processing
--|    of connections means to make the subsystem consistent with its
--|    environment.
--|
--| Parameter Description:
--|    a_subsystem is the subsystem to update
--|
--| Notes:
--|    none
--|*********************************************************************

end Flight_Systems_Connection_Manager;
```

C.15. Separate Procedure body Process_Engine_Connections_To

```
--|*********************************************************************
--| Module Name:
--|    Process_Engine_Connection_To
--|
--| Module Type:
--|    Separate Procedure Body
--|
--| Module Purpose:
--|    Process connections between an engine subsystem and all external
--|        systems.
--|-------------------------------------------------------------------
--| Module Description:
--|    This procedure processes all connections between an engine subsystem
--|        and external systems.  Processing of connections means to make
--|        the subsystem consistent with its environment.
--|
--| Parameter Description:
--|    a_subsystem is the subsystem to update
--|
--| References:
--|    Design Documents:
--|        none
--|
--|    User's Manual:
--|        none
--|
--|    Testing and Validation:
--|        none
--|
--| Notes:
--|    none
```

```
--|-------------------------------------------------------------
--| Modification History:
--|   25Aug87   kl   created
--|
--|-------------------------------------------------------------
--| Distribution and Copyright Notice:
--|   TBD
--|
--| Disclaimer:
--|   "This work was sponsored by the Department of Defense.
--|   The views and conclusions contained in this document are
--|   solely those of the author(s) and should not be interpreted as
--|   representing official policies, either expressed or implied,
--|   of the Software Engineering Institute, Carnegie Mellon University,
--|   the U.S. Air Force, the Department of Defense, or the U.S. Government."
--|*************************************************************

with Ac_Power_Aggregate;
with Integrated_Drive_Object_Manager;
with Engine_Aggregate;
with Rotor2_Object_Manager;
with Standard_Engineering_Types; use Standard_Engineering_Types;

separate (Flight_Systems_Connection_Manager)

procedure Process_Engine_Connections_To (
        A_Subsystem: in Flight_Subsystem_Names.Name_Of_A_Flight_Subsystem) is

    Integrated_Drive_Energy : Integrated_Drive_Object_Manager.Energy;

--
-- A local variable is defined to store the value spark when it is read from
-- the ignition system.  This is a convention, described in the SEI Ada
-- Coding Guidelines (currently under development), to restrict the spread of embedded
--  function calls, i.e., function calls as parameters within other function calls.
--
    Some_Spark : Ignition.Spark;

    function Spark_Conversion (In_Spark : in Ignition_Object_Manager.Spark)
        return Burner_Object_Manager.Spark is
    --|*********************************************************************
    --| Description:
    --|   This function performs a type conversion.  It converts
    --|   the spark from the Ignition to a spark that the
    --|   Burner_Object_Manager can accept.  This is done
    --|   as an example of how the type conversions can be used to
    --|   connect objects which either communicate through a
    --|   valve/regulator, or need different grains of coarseness of
    --|     the information.
    --|   In this case we are assuming that the Igition system
    --|   needs finer information about the spark than does the Burner system.
    --|
    --| Parameter Description:
    --|   In_Spark is the spark that the Ignition supplies.
    --|   return Spark is the spark returned for the Burner
    --|*********************************************************************
    begin
        case In_Spark is
            when 0..2  => RETURN Burner_Object_Manager.None;
            when 3..9  => RETURN Burner_Object_Manager.Low;
            when 10..20 => RETURN Burner_Object_Manager.High;
        end case ;
    end Spark_Conversion;

begin    -- Process_Engine_Connections_To
    --
```

```
-- All engine external connections are handled in this procedure.
-- Each engine has the same kind of connections, but each engine is
-- connected to different instances of other objects.  Thus all engines
-- are handled alike here.  The different connections are described by
-- the engine_aggregate package.
--

--
-- Get_Air_From (the_environment);
-- Give_Air_To (a_diffuser);
--         goes here
--

--
-- Get_Mach_Number_From (the_airframe);
-- Give_Mach_Number_To (a_diffuser);
--         goes here
--

--
-- Get_Discharge_Pressure_From (the_cabin_air);
-- Get_Discharge_Pressure_From (the_air_conditioning_system);
--         any processing of these two pieces of information goes here
-- Give_Discharge_Pressure_To (a_bleed_valve);
--         goes here
--

--
-- Get_Torque_From (the_hydraulic_system);
-- Get_Torque_From (the_oil_system);
-- Get_Torque_From (the_starter_system);
-- Get_Torque_From (the_fuel_system);
-- Get_Torque_From (the_electrical_system);
--         any processing of these five pieces of information goes here
-- Give_Torque_To (a_rotor2);      --          goes here
--
-- For now we are just showing one of these five connections, the one
-- from the electrical system.  For the complete system, all five pieces
-- of information would be gathered and processed before passing the
-- information to the Rotor2.
--
Integrated_Drive_Energy :=
     Integrated_Drive_Object_Manager.Get_Energy_From (
          A_Integrated_Drive => Integrated_Drive_Generators(
               Engines_To_Idg_Map(A_Subsystem))
     );

Rotor2_Object_Manager.Give_Torque_To (
     A_Rotor2 => Engine_Aggregate.Engines(A_Subsystem).The_Rotor2,
     Given_Torque => Torque(Integrated_Drive_Energy)
);

--
-- Get_Fuel_Flow_From (the_fuel_system);
-- Give_Fuel_Flow_To (a_burner);
--         goes here
--

--
-- Get Spark from the Ignition and feed it to the Engine Burner.
--
Some_Spark :=
     Ignition.Get_Spark_From (This_Ignition(Given_Engine_Name));

Burner_Object_Manager.Give_Spark_To (
     A_Burner => Engines(An_Engine).The_Burner,
```

```
        Given_Spark => Spark_Conversion(Some_Spark));
    --

end Process_Engine_Connections_To;
```

C.16. Separate Procedure body Process_Power_Connections_To

```
--|*****************************************************************
--| Module Name:
--|   Process_Power_Connections_To
--|
--| Module Type:
--|   Separate Procedure Body
--|
--| Module Purpose:
--|   process connections between a power subsystem and external systems
--|-----------------------------------------------------------------
--| Module Description:
--|   This procedure processes all connections between a power subystem
--|       and external systems.  Processing of connections means to make
--|       the subsystem consistent with its environment.
--|
--| Parameter Description:
--|   a_subsystem is the subsystem to update
--|
--| References:
--|   Design Documents:
--|     none
--|
--|   User's Manual:
--|     none
--|
--|   Testing and Validation:
--|     none
--|
--| Notes:
--|   none
--|-----------------------------------------------------------------
--| Modification History:
--|   25Aug87   kl   created
--|
--|-----------------------------------------------------------------
--| Distribution and Copyright Notice:
--|   TBD
--|
--| Disclaimer:
--|   "This work was sponsored by the Department of Defense.
--|   The views and conclusions contained in this document are
--|   solely those of the author(s) and should not be interpreted as
--|   representing official policies, either expressed or implied,
--|   of the Software Engineering Institute, Carnegie Mellon University,
--|   the U.S. Air Force, the Department of Defense, or the U.S. Government."
--|*****************************************************************

with Standard_Engineering_Types; use Standard_Engineering_Types;
with Ac_Power_Aggregate;
with Integrated_Drive_Object_Manager;
with Engine_Aggregate;
with Rotor2_Object_Manager;
with Flight_Subsystem_Names; use Flight_Subsystem_Names;

separate (Flight_Systems_Connection_Manager)
```

```
procedure Process_Power_Connections_To(
        A_Subsystem: in Name_Of_A_Flight_Subsystem) is

    Rotor2_Energy: Rpm;

    begin
        case A_Subsystem is
          when Ac_Power =>
                for An_Engine in Engine_1..Engine_4 loop

                    Rotor2_Energy :=
                        Rotor2_Object_Manager.Get_Rpm_From(
                            A_Rotor2 =>
                                Engine_Aggregate.Engines(An_Engine).The_Rotor2
                        );

                    Integrated_Drive_Object_Manager.Give_Energy_To(
                        A_Integrated_Drive =>
                            Ac_Power_Aggregate.Integrated_Drive_Generators
                                Engines_To_Idg_Map(An_Engine)
                        Given_Energy =>
                            Integrated_Drive_Object_Manager.Energy
                                Rotor2_Energy
                        );

                end loop;

            when Dc_Power => null;

            when others => null;

        end case;

end Process_Power_Connections_To;
```

C.17. Package Engine_Updater

```
-- ***************************************************************
-- Module Name:
--   Engine_Updater
--
-- Module Type:
--   Package Specification
--
-- Module Purpose:
--   This package contains the single procedure call to update the
--   simulation of an Engine.  It is the sole interface to the Engines
--   from the perspective of the executive.
-- ---------------------------------------------------------------
-- Module Description:
--   The single operation provided by this package is parameterized with
--   the name of the engine to be updated.  The operation accomplishes
--   two sets of lower-level operations:
--     one to update the state of the objects at the boundries of the
--     engine subsystem which have connections (interfaces) with objects
--     in other subsystems external to the engine subsystem,
--     and another to update all objects internal to the engine subsystem
--     based on the connections (interfaces) between each other.
--   Specifying the name of the engine allows the work to be spead out
--   across the available processing time, and pushes this decision up
--   to a higher, more intelligent being (the executive) to choose the
--   order of updating the engines in the engine subsystem.
--
-- References:
--   Design Documents:
```

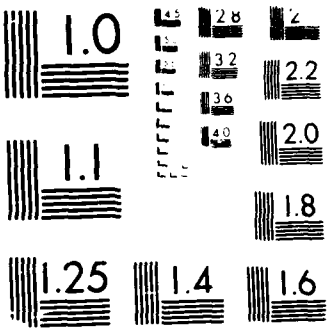
```
--       none
--
--    User's Manual:
--       none
--
--    Testing and Validation:
--       none
--
--  Notes:
--    none
--  ..........................................................
--  Modification History:
--    21Aug87      cpp    created
--  ..........................................................
--  Distribution and Copyright Notice:
--    TBD
--
--  Disclaimer:
--    This work was sponsored by the Department of Defense.
--    The views and conclusions contained in this document are
--    solely those of the author(s) and should not be interpreted as
--    representing official policies, either expressed or implied,
--    of the Software Engineering Institute, Carnegie Mellon University,
--    the U.S. Air Force, the Department of Defense, or the U.S. Government."
--  ************************************************************************

with Flight_Subsystem_Names;    --  Provides the type (definition) of the
use  Flight_Subsystem_Names;    --  names of the engines defined for this
                                --      system:
                                --          Names_Of_A_Flight_Subsystem

package Engine_Updater is

   procedure Update_Engine(Given_Engine_Name: in Name_Of_A_Flight_Subsystem);
   --  ********************************************************************
   --  Description:
   --     Allows the simulation of the Engine Subsystem to be updated
   --          and made consistent. Then other subsystems dependent upon
   --          the Engine Subsystem can access the consistent state of the
   --          Engine Subsystem.  It is an atomic action.
   --
   --  Parameter Description:
   --     Given_Engine_Name
   --           It's type is declared in Flight_Subsystem_Names and is used
   --           to allow a higher, more intelligent being (the executive) to
   --           choose the order of updating the engines in the
   --           engine subsystem.
   --
   --  ********************************************************************

end Engine_Updater;
```

C.18. Package body Engine_Updater

```
--  ************************************************************************
--  Module Name:
--     Engine_Updater
--
--  Module Type:
--     Package Body
--
--  ------------------------------------------------------------------------
```

AD-A191 897 AN OOD (OBJECT-ORIENTED DESIGN) PARADIGM FOR FLIGHT SIMULATORS (U) CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST K J LEE ET AL DEC 87 CMU/SEI-87-TR-43 ESD-TR-87-206 F/G 1/2 2/2 NL

UNCLASSIFIED



```
--| Module Description:
--|   The operation provided by this package allows the "user" to
--|   update the state of an engine, i.e., update the state of the
--|   objects which simulate the individual parts of the engine.
--|
--|   Because this subsystem updater is at the level above the
--|   object managers, we have decided that the subsystem updater
--|   will internally implement the connection manager at this level
--|   since we don't have to go around and touch the object managers
--|   and tell them to update themselves.  The object managers update
--|   themselves (their state) when the connection is made and the
--|   state information is given to them.
--|
--|
--|
--| References:
--|     Design Documents:
--|       Engine Physical Model Diagram.
--|
--|     Testing and Validation:
--|       none
--|
--| Notes:
--|   THIS IS NOT A FULL IMPLEMENTATION!!!
--|   The code is done to demonstrate the process of connecting objects
--|   in a subsystem.
--|
--|     The connection manager wasn't implemented at this level
--|   for the reasons stated above in the Module Description.
--|
--|   Once the Engine subsystem has been updated, i.e., its state
--|   made consistent, any object whose state is needed by objects
--|   in other subsystems can be had by directly accessing the object
--|   and getting its state.
--|
--|   All internal routines preform a type conversion on the data
--|   when the data is transferred from engine object to engine object.
--|   This is done to allow flexibility and greater potential for reuse
--|   of object managers.  Another reason for type conversions, which
--|   is related to the flexibility issue, is that there may be something
--|   to model at the connection between the objects, i.e., a valve,
--|   regulator, etc., for which an object manager is not necessary.
--|   Therefore, any calculations or transformations which need to occur
--|   and be modelled at the connection can be made when the connection
--|   between the objects occur.
--|
--|-------------------------------------------------------------------
--| Modification History:
--|   24Aug87       cpp    created
--|
--|-------------------------------------------------------------------
--| Distribution and Copyright Notice:
--|   TBD
--|
--| Disclaimer:
--|   "This work was sponsored by the Department of Defense.
--|   The views and conclusions contained in this document are
--|   solely those of the author(s) and should not be interpreted as
--|   representing official policies, either expressed or implied,
--|   of the Software Engineering Institute, Carnegie Mellon University,
--|   the U.S. Air Force, the Department of Defense, or the U.S. Government."
--|*******************************************************************


with Standard_Engineering_Types;
with Engine_Aggregate;              --| Provides the type:
```

```
                        --|        Engine_Representation
                        --|      Provides the object which allows
                        --|      us to specify engine parts:
                        --|        Engines
with Diffuser_Object_Manager;     --| Provides the type:
                        --|
                        --| Provides the function:
                        --|
with Rotor1_Object_Manager;       --| Provides the type:
                        --|
                        --| Provides the procedure and function:
                        --|
                        --|
with Fan_Duct_Object_Manager;         --| Provides the type:
                        --|
                        --| Provides the procedure:
                        --|
with Rotor2_Object_Manager;       --| Provides the type:
                        --|
                        --| Provides the procedure and function:
                        --|
                        --|
with Burner_Object_Manager;       --| Provides the type:
                        --|
                        --| Provides the procedure and function:
                        --|
                        --|
with Exhaust_Object_Manager;      --| Provides the type:
                        --|
                        --| Provides the procedure:
                        --|
package body Engine_Updater is

  procedure Update_Engine(Given_Engine_Name: in Name_Of_A_Flight_Subsystem) is
  --|*********************************************************************
  --| Description:
  --|   Allows the simulation of the Engine Subsystem to be updated
  --|       and made consistent. Then other subsystems dependent upon
  --|       the Engine Subsystem can access the consistent state of the
  --|       Engine Subsystem. It is an atomic action. The user must
  --|       specify the engine to be updated.
  --|
  --|       The object managers which simulate the various parts of the
  --|       engine, thus comprising the engine subsystem, are
  --|       needed to update the subsystems state are the following:
  --|         Diffuser_Object_Manager
  --|         Rotor1_Object_Manager
  --|         Fan_Duct_Object_Manager
  --|         Rotor2_Object_Manager
  --|         Burner_Object_Manager
  --|         Exhaust_Object_Manager
  --|       The connections between these objects and the state information
  --|       flowing between the objects were derived solely from the
  --|       Engine Physical Model Diagram shown in SEI Technical Report #CMU/SEI-87-TR-43,.
  --|       An OOD Paradigm for Flight Simulators
  --|
  --| Parameter Description:
  --|   Given_Engine_Name
  --|         It's type is declared in Engine_Names and is used to allow
  --|         a higher, more intelligent being (the executive) to
  --|         choose the order of updating the engines in the
  --|         engine subsystem.
  --|
  --|    Note:
  --|       This routine models the connection manager for this level.
  --|*********************************************************************
```

```
    Diffuser_Discharge_Pressure     : Standard_Engineering_Types.Pressure;
    Diffuser_Discharge_Temperature: Standard_Engineering_Types.Temperature;
    Diffuser_Discharge_Air_Flow     : Standard_Engineering_Types.Air_Flow;
  begin
  --
  -- Model the connection characterized by the dependence of the Rotor1
  -- on the Diffuser for Pneumatic_Energy.
  --
  -- NOTE, no type conversion is necessary because both types are based
  -- on Standard_Engineering_Types Package definitions.
  --
    Diffuser_Object_Manager.Get_Discharge_Air_From(
      A_Diffuser => Engine_Aggregate.Engines(Given_Engine_Name).The_Diffuser,
      Returning_Discharge_Pressure  =>    Diffuser_Discharge_Pressure,
      Returning_Discharge_Temperature => Diffuser_Discharge_Temperature,
      Returning_Discharge_Air_Flow  =>    Diffuser_Discharge_Air_Flow
    );
    Rotor1_Object_Manager.Give_Fan1_Inlet_Air_To(
      A_Rotor1  => Engine_Aggregate.Engines(Given_Engine_Name).The_Rotor1,
      Given_Fan1_Inlet_Pressure     =>    Diffuser_Discharge_Pressure,
      Given_Fan1_Inlet_Temperature  =>    Diffuser_Discharge_Temperature,
      Given_Fan1_Inlet_Air_Flow     =>    Diffuser_Discharge_Air_Flow
    );
  --
  -- .
  -- .
  -- .
  --


  end Update_Engine;

end Engine_Updater;
```

C.19. Package Engine_Aggregate

```
--|*************************************************************************
--| Module Name:
--|   Engine_Aggregate
--|
--| Module Type:
--|   Package Specification
--|
--| Module Purpose:
--|   This package names the TurboRotor1 Engines and their parts.
--|-------------------------------------------------------------------------
--| Module Description:
--| A TurboRotor1 Engine is an aggregate of parts:
--|   Diffuser,
--|   Rotor1,
--|   Fan_Duct,
--|   Rotor2,
--|   Bleed_Valve,
--|   Burner,
--|   Exhaust.
--|
--| The parts of a TurboRotor1 Engine are objects which have state.
--| Each part is managed by it's own object
--| manager. This package builds the four engines by calling on
--| the various object managers to create the parts. It then stores
--| references to the parts in a constant array indexed by the
--| Name_Of_A_Flight_Subsystem which is taken from the
--| Flight_Subsystem_Names package. The constant array
```

```
--|  is created when the package is elaborated.  The constant array is
--|  called Engines.  A part of an Engine is referenced as:
--|     Engines(Engine_Name).The_<part_kind>
--|  For example, the Rotor1 of the second Engine is:
--|     Engines(Engine_2).The_Rotor1
--|
--|
--| References:
--|    Design Documents:
--|       none
--|
--|    User's Manual:
--|       none
--|
--|    Testing and Validation:
--|       none
--|
--| Notes:
--|  Optimizations which were implemented: the initialization of Engines
--|  occurs at the declaration of the Object instead of the body because
--|  the number of engines and the parts shouldn't change; thus the object
--|  was also made a constant array of Engines.
--|
--|-------------------------------------------------------------------
--| Modification History:
--|    20Apr87        cpp    created
--|
--|-------------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    TBD
--|
--| Disclaimer:
--|  "This work was sponsored by the Department of Defense.
--|  The views and conclusions contained in this document are
--|  solely those of the author(s) and should not be interpreted as
--|  representing official policies, either expressed or implied,
--|  of the Software Engineering Institute, Carnegie Mellon University,
--|  the U.S. Air Force, the Department of Defense, or the U.S. Government."
--|*******************************************************************

with Flight_Subsystem_Names;    --| Provides the engine names to
use  Flight_Subsystem_Names;    --| create instances of the engines
                                --| in the system.
with Diffuser_Object_Manager;   --| Provides the private type
use  Diffuser_Object_Manager;   --| Diffuser and a function to
                                --|     create a New_Diffuser.
with Rotor1_Object_Manager;     --|  Provides the private type
use  Rotor1_Object_Manager;     --|  Rotor1 and a function
                                --| to create a New_Rotor1.
with Fan_Duct_Object_Manager;       --| Provides the private type
use  Fan_Duct_Object_Manager;   --| Fan_Duct and a function
                                --| to create a New_Fan_Duct.
with Rotor2_Object_Manager;     --|  Provides the private type
use  Rotor2_Object_Manager;     --|  Rotor2 and a function
                                --| to create a New_Rotor2.
with Bleed_Valve_Object_Manager;  --| Provides the private type
use  Bleed_Valve_Object_Manager;  --| Bleed_Valve and a function
                                --| to create a New_Rotor2.
with Burner_Object_Manager;     --|  Provides the private type
use  Burner_Object_Manager;     --|  Burner and a function
                                --| to create a New_Burner.
with Exhaust_Object_Manager;    --|  Provides the private type
use  Exhaust_Object_Manager;    --|  Exhaust and a function
                                --| to create a New_Exhaust.
package Engine_Aggregate is
```

```
   type Engine_Representation is        -- Defines an engine representation
   record                               -- as consisting of:
     The_Diffuser : Diffuser;
     The_Rotor1   : Rotor1;
     The_Fan_Duct    : Fan_Duct;
     The_Rotor2   : Rotor2;
     The_Bleed_Valve  : Bleed_Valve;
     The_Burner   : Burner;
     The_Exhaust  : Exhaust;
   end record ;

   -- Define an object which holds all 4 engines in the system and
   -- initialize them (i.e., all their parts).
   Engines: constant array (Engine_1..Engine_4) of Engine_Representation :=
     (Engine_1  => (
        The_Diffuser   =>    New_Diffuser,
        The_Rotor1     =>    New_Rotor1,
        The_Fan_Duct   =>    New_Fan_Duct,
        The_Rotor2     =>    New_Rotor2,
        The_Bleed_Valve => New_Bleed_Valve,
        The_Burner         =>    New_Burner,
        The_Exhaust        =>    New_Exhaust
        ),
      Engine_2  => (
        The_Diffuser   =>    New_Diffuser,
        The_Rotor1     =>    New_Rotor1,
        The_Fan_Duct   =>    New_Fan_Duct,
        The_Rotor2     =>    New_Rotor2,
        The_Bleed_Valve => New_Bleed_Valve,
        The_Burner         =>    New_Burner,
        The_Exhaust        =>    New_Exhaust
        ),
      Engine_3  => (
        The_Diffuser   =>    New_Diffuser,
        The_Rotor1     =>    New_Rotor1,
        The_Fan_Duct   =>    New_Fan_Duct,
        The_Rotor2     =>    New_Rotor2,
        The_Bleed_Valve => New_Bleed_Valve,
        The_Burner         =>    New_Burner,
        The_Exhaust        =>    New_Exhaust
        ),
      Engine_4  => (
        The_Diffuser   =>    New_Diffuser,
        The_Rotor1     =>    New_Rotor1,
        The_Fan_Duct   =>    New_Fan_Duct,
        The_Rotor2     =>    New_Rotor2,
        The_Bleed_Valve => New_Bleed_Valve,
        The_Burner         =>    New_Burner,
        The_Exhaust        =>    New_Exhaust
        )
   );

end Engine_Aggregate;
```

C.20. Package System_Power_Updater

```
--| ***************************************************************
--| Module Name:
--|    System_Power_Updater
--|
--| Module Type:
--|    Package Specification
--|
```

```
--|-------------------------------------------------------------
--| Module Description:
--|    stub package specification for completion of the Engine system
--|
--| References:
--|    Design Documents:
--|       none
--|
--|    Testing and Validation:
--|       none
--|
--| Notes:
--|    none
--|-------------------------------------------------------------
--| Modification History:
--|    21Aug87   kl   created
--|
--|-------------------------------------------------------------
--| Distribution and Copyright Notice:
--|    TBD
--|
--| Disclaimer:
--|    "This work was sponsored by the Department of Defense.
--|    The views and conclusions contained in this document are
--|    solely those of the author(s) and should not be interpreted as
--|    representing official policies, either expressed or implied,
--|    of the Software Engineering Institute, Carnegie Mellon University,
--|    the U.S. Air Force, the Department of Defense, or the U.S. Government."
--|*************************************************************

with Flight_Subsystem_Names;

package System_Power_Updater is

procedure Update_System_Power (
    A_Subsystem: in Flight_Subsystem_Names.Name_Of_A_Flight_Subsystem);
--|*************************************************************
--| Description:
--|    Allows the simulation of the Electrical Subsystem to be updated
--|      and made consistent.
--|
--| Parameter Description:
--|    a_subsystem is the subsystem to update
--|*************************************************************

end System_Power_Updater;
```

References

[1] Booch, Grady.
Software Engineering with Ada.
The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

[2] Booch, Grady.
Software Components with Ada.
The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

[3] Hesse, Walter J. and Mumford, Nicholas V. S., Jr.
Jet Propulsion for Aerospace Applications.
Pitman Publishing Corporation, New York, NY, 1964.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-87-TR-43	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-87-206
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INSTITUTE 6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE
6c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JOINT PROGRAM OFFICE 8b. OFFICE SYMBOL (If applicable) SEI JPO	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962885C0003
8c. ADDRESS (City, State and ZIP Code) CARNEGIE MELLON UNIVERSITY SOFTWARE ENGINEERING INSTITUTE JPO PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. / PROJECT NO. N/A / TASK NO. N/A / WORK UNIT NO. N/A
11. TITLE (Include Security Classification) AN OOD PARADIGM FOR FLIGHT SIMULATORS	
12. PERSONAL AUTHOR(S) KEN LEE, ET AL	
13a. TYPE OF REPORT FINAL 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) DECEMBER 1987 15. PAGE COUNT 106
16. SUPPLEMENTARY NOTATION	
17. COSATI CODES FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) OBJECT-ORIENTED, SOFWTARE ENGINEERING, ADA, FLIGHT SIMULATORS

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

THIS REPORT PRESENTS A PARADIGM FOR OBJECT-ORIENTED IMPLEMENTATIONS OF FLIGHT SIMULATORS. IT IS A RESULT OF WORK ON THE ADA SIMULATOR VALIDATION PROGRAM (ASVP) CARRIED OUT BY MEMBERS OF THE TECHNICAL STAFF AT THE SOFTWARE ENGINEERING INSTITUTE (SEI).

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ OTIC USERS ☒	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL SHINGLER	22b. TELEPHONE NUMBER (Include Area Code) (412) 268-7630 22c. OFFICE SYMBOL SEI JPO

END
DATE
FILMED
5-88
DTIC